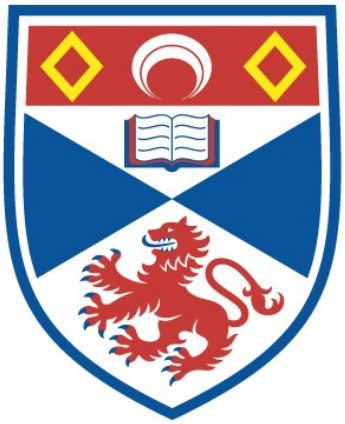
THE IMPERATIVE IMPLEMENTATION OF ALGEBRAIC DATA TYPES

Muffy Thomas

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews



1988

The Imperative Implementation of Algebraic Data Types

by
Muffy Thomas

A thesis submitted for the degree of Doctor of Philosophy
in the University of St. Andrews

1987

Abstract

The synthesis of imperative programs for hierarchical, algebraically specified abstract data types is investigated. Two aspects of the synthesis are considered: the choice of data structures for efficient implementation, and the synthesis of linked implementations for the class of ADTs which insert and access data without explicit key. The methodology is based on an analysis of the algebraic semantics of the ADT.

Operators are partitioned according to the behaviour of their corresponding operations in the initial algebra. A family of relations, the *storage relations* of an ADT, is defined. They depend only on the operator partition and reflect an observational view of the ADT. The storage relations are extended to *storage graphs*: directed graphs with a subset of nodes designated for efficient access.

The data structures in our imperative language are chosen according to properties of the storage relations and storage graphs. Linked implementations are synthesised in a stepwise manner by implementing the given ADT first by its storage graphs, and then by linked data structures in the imperative language. Some circumstances under which the resulting programs have constant time complexity are discussed.

Declaration

I Muffy Thomas hereby declare that this thesis has been composed by myself, that it is a record of my own work, and that it has not been accepted in partial or complete fulfillment of any other degree or professional qualification.

Signed **Date** 20.12.87

I was admitted to the Faculty of Science of the University of St. Andrews under Ordinance General No. 12 in October 1980 and as a candidate for the degree of Ph.D in November 1981.

Signed **Date** 20.12.87

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate to the Degree of Ph. D.

Signature of Supervisor **Date**

21 xii 87

In submitting this thesis to the University of St. Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker.

Acknowledgements

I would like to thank Roy Dyckhoff for his supervision. His patient and constructive guidance has been a great help to me; I could not have finished this thesis without his encouragement. I also thank Don Sannella for acting as supervisor when Roy was away on sabbatical.

I am grateful to Jeremy Dick for introducing me to term rewriting, and Klaus Jantke for introducing me to inductive inference. Discussions with each of them have helped me to develop many of the ideas contained in this thesis.

This thesis was completed when I was working in the Computing Science Department at Stirling University; the Department has provided resources and a good environment in which to work. I thank my colleagues at Stirling: Simon Finn, Alan Hamilton, Simon Jones, Maurice Naftalin, and Chic Rattray, for many helpful and friendly discussions. I particularly thank Alan for his comments on a draft of the thesis, Chic for many years of encouragement, and Maurice for being such an agreeable office-mate.

Finally, I thank the institutions which have provided financial support: St. Andrews University, the Committee of Vice-Chancellors and Principals, the Institute of Chartered Secretaries and Administrators, and the Hilda Martindale Trust.

Contents

Chapter Three: Storage Relations

Chapter Four: Specifying Storage Relations

Chapter Five: Implementation and Storage Graphs

Chapter Six: Implementation by Imperative Language

Chapter Seven: Conclusions and Future Work

Chapter One
Specification and Implementation

1.1 Introduction

Algebraic specification is an axiomatic abstraction technique which encourages the construction of correct and efficient programs by separating the two concerns of *specification* and *implementation*; specification is concerned with *what* whereas implementation is concerned with *how*.

The specifier concentrates on problem solving and capturing the intended behaviour of the data objects as an abstract data type, or an *ADT*. Informally, an ADT specification consists of three components: a linguistic component, an assertive component containing the set of logical sentences describing the properties of whatever is being specified, and a deductive component containing the rules of inference. When a specification is in some sense "good", (perhaps when properties such as consistency and completeness are satisfied) then it may be implemented.

The implementer concentrates on the problems of *efficient* representation in the implementation language whilst ensuring that the implementation is *correct*. Correctness ensures that the meaning of the specification is preserved; the implementation must *satisfy* the given axioms. An implementation language may be either directly executable by a machine, or in the spirit of step-wise refinement, it may be closer to an executable language than the specification language. The efficiency of an implementation is measured by its space and time complexity and so when constructing the implementation, the implementer must be aware of the time and storage requirements of each of the constructs of the implementing language.

In this thesis we consider some of the problems of implementing *equationally* specified hierarchical ADTs [EhM 85], [ADJ 78] *efficiently* in an imperative programming language such as Pascal.

An example will highlight some of the issues involved. Consider the following "mystery" specification:

```
spec        Mystery

sorts       nat, thing

ops         0          : nat
            succ       : nat -> nat
            error      : nat
            one        : thing
            two        : thing nat -> thing
            three      : thing  -> thing
            four       : thing -> nat
eqns
∀d:nat.                 four(two(one,d)) =  d
∀x:thing,d,d':nat.      four(two(two(x,d),d')) = four(two(x,d))
                        three(one) =  one
                        four(one) = error
∀d:nat.                 three(two(one,d)) =  one
∀x:thing,d,d':nat.      three(two(two(x,d),d'))=two(three(two(x,d)),d')
end
```

How might we implement such a specification in Pascal?

First, we decide how operations and sorts are implemented: by functions and values, or by imperative procedures and variables. For the moment, we assume the former for the primitive sort and primitive-sorted operations (namely *nat* and the operations *succ* and *four*), and the latter for the hierarchical sort and hierarchical-sorted operations (namely *thing* and operations *one*, *two* and *three*). We assume that an implementation for the elements of sort *nat* is given elsewhere.

Second, we choose data structures to represent the elements of sort *thing* and we define the representation mapping between *thing* and the chosen data structures. Do we choose arrays and indices, or one of the many linked structures such as singly-linked lists, doubly-linked lists, or binary trees? This decision must depend on determining what kind of *structure*, or *behaviour*, **Mystery** specifies and which of its properties are most important and must be represented efficiently. The crucial step seems to be to develop an understanding (i.e. to build a model) of the kind of behaviour specified by **Mystery**; but how can we determine that behaviour from this equational specification?

Third, using the representation mapping, we define the procedures and functions which correctly implement the operations *one*, *two*, *three* and *four* using the chosen data structures.

Before considering these decisions any further, perhaps if we rename the sorts and operators of **Mystery** we will recognise the specification of a familiar behaviour:

```
spec        Queue

sorts       nat, queue

ops         0          : nat
            succ       : nat -> nat
            error      : nat
            eq         : queue
            add        : queue nat -> queue
            front      : queue -> nat
            dequeue    : queue  -> queue
eqns
∀d:nat.                 front(add(eq,d)) =  d
∀q:queue,d,d':nat.      front(add(add(q,d),d')) = front(add(q,d))
                        dequeue(eq) =  eq
                        front(eq) = error
∀d:nat.                 dequeue(add(eq,d)) =  eq
∀q:queue,d,d':nat.
          dequeue(add(add(q,d),d'))=add(dequeue(add(q,d)),d')
end
```

We know from our own experience and from texts such as [Mar 86] that **Queue** is a *linear* data type. On this basis we might choose an array with two indices as a representing data structure:

```
type Queue = record
     s: array[1..max] of integer;
     front, rear: integer
     end;
```

We know that two indices are necessary because insertions and deletions (i.e. the operations specified by *add* and *dequeue*) occur at both ends of the linear structure; this also implies that in order to use the whole array, modular arithmetic is required. We note that an array-based implementation introduces the restrictions of size bounds and so a notion of implementation correctness must take these bounds into account. As an example, assuming that `max` is a predefined constant and `Overflow` is a predefined exception-procedure, we give the following procedure `ADD` as an implementation of the operation *add*:

```
procedure ADD(var q:Queue; n:integer);
begin
with q do
   if rear=max then rear:=1 else rear:=rear+1;
   if rear=front then Overflow else q[rear]:=n
end;
```

Alternatively, we might choose a singly-linked list as the representing data structure:

```
type List = record
     item: integer;
     next: ^List
     end;
```

It is easy to see that the choice of this data structure and a straightforward representation mapping will cause either the implementation of *add*, or the implementations of *dequeue* and *front*, to traverse the entire list. Thus, either the implementation of *add*, or the implementations of *dequeue* and *front*, will have a time complexity which is proportional to the size of the queue. For example, if we allow the head of the list to represent the least recent addition to the queue, then the following procedure ADD implements the operator *add*:

```
procedure ADD(var q:List; n:integer);
var p,t : ^List;
begin
new(p);  p^.item:=n;  p^.next:=nil;
if q.next = nil then q.next:=p
else  begin  t:=q.next;
             while t^.next<>nil do t:=t^.next; t^.next:=p
      end
end;
```

Still keeping this choice of data structure, a more complex representation which exploits the fact that the last element of a singly-linked list does not have an outgoing pointer enables us to implement all the operations in constant time. If we allow the head of the list to represent the most recent addition to the queue, and we link this element to the least recent addition to the queue, then although the front of the queue is not immediately accessible, we can always reach the front or the rear of the queue in constant time.

Finally, a more (time) efficient alternative is to allocate more space to the data structure and keep pointers to both the front and the rear of the linked list:

```
type Queue = record
     front, rear: ^List
     end;
```

Using a straightforward representation mapping: the head of the list represents the least recent addition to the queue and the last element of the list represents the most recent addition to the queue, we can now give constant time implementations (using linked data structures) for all the operations. For example, the ADD procedure is:

```
procedure ADD(var q:Queue; n:integer);
var p : ^List;
begin
new(p);  p^.item:=n;  p^.next:=nil;
with q do
  begin if front=nil then front:=p else rear^.next:=p;
  rear:=p
   end
end;
```

This brief example illustrates some of the issues considered in this thesis: namely, the choice of data structures and representation mappings and the effects of these decisions on the time and space complexity of the implementation.

We consider imperative languages because although functional programming languages may be more attractive, imperative languages are still widely used and we believe that this situation will probably continue for some time. Imperative programs are desirable because they run efficiently on conventional computers. They are undesirable because it is very difficult to reason about an imperative program. We particularly encourage the specification of imperative programs because these undesirable aspects disappear when the programs are derived from formal specifications: we can reason about the programs at the abstract level of the specification instead of at the imperative level.

Students and programmers have been implementing ADTs in imperative languages during the last decade according to intuition and informal rules. There is little methodology and there are few software tools available to aid the implementation of ADTs

correctly and efficiently; in particular, the choice of implementing data structures has largely been ignored. Instead, much of the research in the past has concentrated on methodologies for constructing specifications [GHM 78a], [PeV 78], [GeM 86], implementating specifications by functional programs [Moi 82], [Pro 86], and specification languages [GoB 83], [Wir 86].

The topic of this thesis arose from work at Stirling University on the SERC Alvey project number 007: "An Automatic Programming Expert for Implementing Specifications as Quality Structured Programs". It became apparent early on in the project that a knowledge-based approach consisting of encoding the "programming knowledge" contained in such standard texts as [AHU 83] and [Mar 86] would not be suitable for a formal subject such as program specification and transformation. A formal methodology seems to be crucial for a project whose aim is to analyse and where possible, automatically implement ADTs. In this thesis we have concentrated on formalising the methodology with the intention of enabling the automatic synthesis of imperative programs in the future.

The aim of the thesis is twofold. First, we formalise some aspects of choosing the implementing data structures and representation mappings which lead to efficient implementations for the class of hierarchical ADTs which do not insert or retrieve data by explicit key. We refer to these ADTs as *keyless* and *implicitly keyed* ADTs. Second, we provide a method for choosing linked data structures and constructing correct and efficient imperative implementations using those data structures.

1.2 Issues in Specification and Implementation

In this section we survey some of the issues of specification and implementation. We put into context the issues which we consider and mention some of the issues which we have not considered.

1.2.1 Referential Transparency and Algorithm Analysis

The example implementations given in the introduction have destroyed the referentially transparent aspects of the specification. Namely, when `q` is a variable of type `Queue`, then the queue *represented* by `q` after the execution of the procedure call

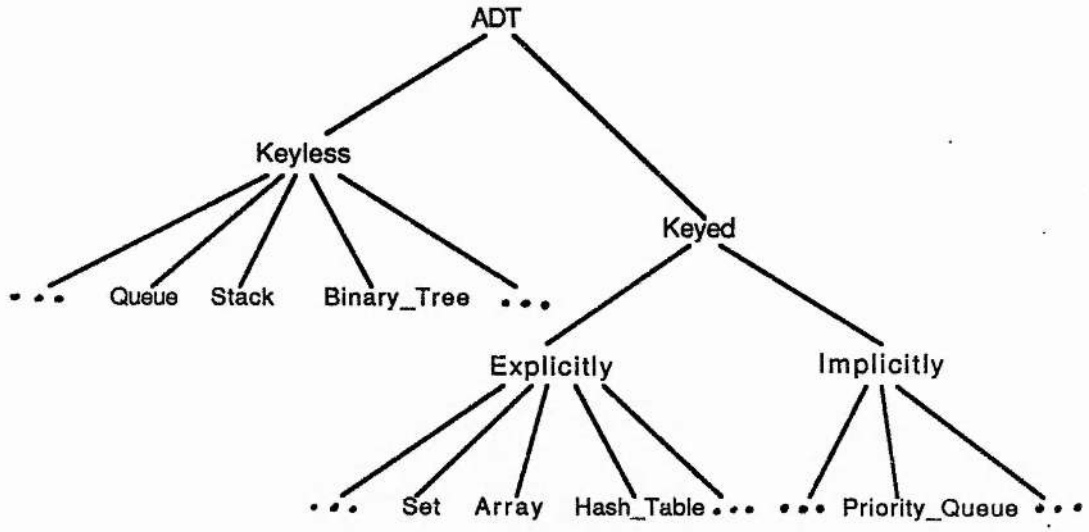
ADD (q, 3) is different from the queue which q *represented* before execution. Although the *value* of q is unchanged, (i.e. the location), the queue which q *represents* has changed and we have lost the property of substitution.

Informally, the procedure ADD may only be considered as an implementation for the operator add when the substitution property is not required in the context in which ADD occurs. The absence of the substitution property in an algorithm allows us to replace function calls by (more efficient) procedure calls. Namely, if we can replace the value of a variable x by f (x) ; i.e. we can destructively update x with the command x:=f (x) , and we have a procedure P which fulfills the specification {true} P (x) {x'=f (x)}, then we can replace x:=f (x) by P (x) . (Of course P generalises to procedures with any number of parameters.) A procedure can still (inefficiently) implement an operator even when the substitution property is required: when x cannot be replaced by f (x) , then we can make a copy of x, in variable z say, call P (z) , and then refer to the value of z.

In the following, we do not consider the algorithms which use the ADTs and so we do not attempt to deduce when a data structure can be destructively updated. Moreover, we do not consider information such as the context and frequency of an operation in an algorithm. These topics are pursued elsewhere; for example, they are discussed in [JøS 86], [Sch 85] and [Sch 86]. Our aim is to define procedures, as described above, for the hierarchical operations, and (referentially transparent) functions for the primitive operations.

1.2.2 A Taxonomy of ADTs

Most standard texts on data structures (for example, [AHU 83] and [Mar 86]) informally group together data structures with similar structure. Here, we group together ADTs according to similar criteria. The figure below displays a taxonomy of ADTs:



A keyless ADT imposes a structure on the set of elements of the primitive sort which is *independent* of any prior relationship between the elements. Storage and retrieval must therefore be specified by reference to some chronological ordering.

A keyed ADT imposes a structure on the set of elements of the primitive sort which is *dependent* on some prior property of the elements. If the key is explicit, then storage and retrieval are specified by reference to some relation between indices and primitive elements. If the key is implicit, then storage and retrieval are specified by reference to some ordering relation between primitive elements.

The way in which an ADT is implemented depends on whether and how the ADT uses keys. Whereas efficient implementations of keyless ADTs may, in general, be constructed without reference to the algorithm which uses the ADT, the efficiency of implementations of keyed ADTs is almost completely dependent upon the dynamic use of the ADT. For example, the efficient implementation of these latter ADTs depends on the density of the data, the frequency of access to particular items and the order of access. We shall concentrate on the problem of choosing representations for keyless and implicitly keyed ADTs. An explicitly keyed ADT such as Set is usually a standard type in constructive specification languages such as VDM [Jon 86]. The choice of efficient representations for sets in constructive specifications has been studied in [SSS 81] and [PaK 83].

1.2.3 Imperative Data Structures and Implementations

Hierarchical ADTs are implemented in a hierarchical way. We assume that implementations of the primitive data types, (the "data"), are given elsewhere and concentrate on choosing the data structures which represent the hierarchical sort.

Data structures in languages such as Pascal are classified by storage allocation mechanism: static or dynamic.

Array-based implementations exploit the fact that the storage is allocated sequentially and the index type is ordered. When arithmetic operations are also defined in the index type, then related data items may be stored at positions, or cells, in the store whose difference is defined by the corresponding arithmetic expression. In contrast, dynamic, or linked implementations, explicitly link cells together. Because the positions of free cells are not related (they are removed one at a time from the heap at runtime), cells containing related items must be explicitly linked together. Linked implementations use only as much space as is needed (apart from the overhead of links). Static implementations may either waste cells or provide an insufficient number of them; however, they may be preferable when specifications are bounded, or when efficient, random access to the data is required.

A common programming paradigm is to build a new data structure from the structure containing the data, (the *implementation structure*), and the address, or index, of one or more of the positions in the *implementation structure*. These positions are referred to as *entry points*; this paradigm will be adopted when choosing representations of ADTs. The number and nature of the entry points can affect the efficiency of the operations on the data structure. We have seen, from the implementation of the `Queue` example in the introduction, that the allocation of an additional *entry point* can reduce the time complexity of an operation. There are no fixed rules about how to choose *entry points*. For an efficient implementation, the choice must depend on both the underlying *implementation structure* and the intended operations on the data structure.

1.2.4 Error-Handling

The explicit handling of errors and exceptions is avoided by using a specification language with partially-ordered sorts and by imposing some constraints on the

specifications. Error handling is a large topic and it is considered elsewhere. For example, explicit error-handling using partially-ordered sorts is discussed in [Gog 87], and [BBG 84] gives a method for constructing programs with exception handling.

The remainder of this chapter contains an overview of the thesis and a review of the related work.

1.3 Overview of Thesis

In chapter 2, we define some notation for the class of hierarchical, keyless and keyed ADTs, and a partition on the operations of keyless and implicitly keyed ADTs. The partition groups together operations with similar behaviour; this will allow us to consider keyless/implicitly keyed ADTs in general.

In chapter 3, the *storage relations* of an ADT are defined. These are binary relations on the initial algebra and they depend only on the operator partition. They describe an "observational view" (with respect to the data) of how the data is stored, retrieved, and manipulated in the ADT. ADTs are classified according to the properties of the *storage relations*; the properties of the *storage relations* of an ADT are referred to as the *storage type*.

In chapter 4, we discuss how the *storage relations* can be presented syntactically as a specification.

In chapter 5, the *storage relations* are extended to *storage graphs*: *storage relations* with information about which nodes should be efficiently accessible at all times. These nodes are called *access nodes*; they depend on elements of the ADT and they are defined by *access operations*. We give two strategies for determining the *access operations*: the first is a general, automatic strategy whereas the second may require some intervention because it involves the implementation of the given ADT by its *storage relations*.

In chapter 6, we discuss how to choose the implementing data structures according to the properties of the *storage relations* and *storage graphs*. The *implementing structure* is determined by the *storage type* and the *entry points* are determined by the *access operators*. We give a strategy for implementing ADTs in an imperative language using linked data structures and analyse the efficiency of the resulting implementations.

Finally, chapter 7 contains a summary, our conclusions, and plans for future work.

1.4 Related Work

In this section some of the literature relevant to the problems of automating the implementation of ADTs is surveyed. Literature which is relevant only to the topic of a particular chapter will be discussed in that chapter. Here, we consider the topics of choosing representations, evaluating the efficiency of representations, and constructing implementations. In the following section we draw some conclusions about the related work and discuss the contribution of this work within this context.

We consider pre-specification, constructive and algebraic specification techniques, equational, logical, functional and imperative style implementation languages, and knowledge-based, analytical, and transformational approaches. We begin with a brief review of some of the early, and perhaps neglected, works on data structures before the advent of abstract data types and formal specification.

1.4.1 Pre-Specification

The ideas of representing data structures with graphical notations was first suggested nearly twenty years ago in [DIm 69], [Ear 71], and [Ros 71].

In [DIm 69], D'Imperio distinguishes between *data structures*, data elements and the (computer-independent) relationships between them, and *storage structures*, the computer storage "slots" and their physical or logical adjacency. Data structures are described somewhat informally using *diagrams* (a graphical notation with several kinds of nodes and edges) and tables. Storage structures are also described using diagrams. A storage structure is interpreted as the image of a data structure in computer memory and so it describes the relationships between the storage slots at quite a low level. For example, one kind of edge represents accessibility in one addressing cycle whereas another kind represents accessibility in several cycles. In [Ear 71], Earley suggests that data structures are directed graphs, and a programming language called *VER* which implements digraphs is proposed. Also around the same time, [Ros 71] defines a data

structure as a collection of primitive items along with a set of relations on the items. Each data structure is represented as a directed graph and the representation is called a *data graph*. Data graphs must fulfill certain properties; for example, they must be strongly-connected. The problem of how data graphs can be realised, or implemented, in store is formalised but efficiency is not considered.

A programming language based on Rosenberg's data graphs, DSDML, (Data Structure Description and Manipulation Language) is proposed in [ShS 74]. Arbitrary graphs cannot be constructed in the language as the operations are biased toward acyclic trees and linear lists. The interesting aspect of this language is that in order to control access to nodes in the graphs, there is a division between *static* and *dynamic* nodes. Static nodes are the headers, or entry points, whereas dynamic nodes are the nodes which can only be accessed by following non-trivial paths from the static nodes. Static nodes cannot be deleted but the dynamic nodes may be created and deleted by the graph operations.

1.4.2 Constructive Specification

Much of the related literature concerning the automation of the implementation of ADTs refers to constructive specifications. Most of the work was carried out before the advent of the standard constructive specification languages such as Z [Hay 87] and VDM [Jon 86], and the formal approaches to data refinement. The literature can be classified, rather broadly, according to the way in which the operations are specified: by imperative algorithm, by assertions, and by pre- and post-conditions.

[Low 78] and [KaB 81] consider the implementation of specifications which use imperative constructs such as assignment, iteration, and gotos. Their approach is interactive and knowledge-based. The specifications are analysed for various algorithmic properties such as frequency of operations, range of operations, and size bounds. The choice of representation is made according to cost functions which depend on these properties. The choice is made from a library of representations; for example, [Low 78] uses a library containing 8 possible representations for sets. Neither approach includes a formal semantics, nor definitions and proofs that their implementations are correct.

[RoK 77] and [RoT 78] consider the choice of representations for specifications which already contain descriptions of structural properties. [RoK 77] explicitly excludes the possibility of considering algebraic specifications because "it would be quite

difficult for an automatic selection system to reason about an algebraic specification". Both approaches are knowledge-based and implementations are defined in terms of locations and addresses. It is not obvious that either approach has a formal semantics. The system of [RoK 77] is an interactive "expert" system involving both automatic program synthesis and simple instantiation of fixed structures based on numerical information. It contains a knowledge base of ADTs such as Set, Tree, and Sequence, along with possible implementations and cost functions. An ADT is specified by assertions such as *has-ordering* and *must-be-sorted*, and the system tries to find an efficient representation by attempting to view the ADT as a combination of the types in the library. [RoT 78] is a little more rigorous and ADTs are specified by binary relations with properties such as replication, ordering, degree (1:1, 1:many, many:1), connectivity, and access by structural property or by key. The system attempts to match the specification with an existing one in the library; if this fails then it attempts to generate an implementation.

To conclude this section, we mention some of the recent work concerning the implementation of VDM-style pre-, post-condition specifications.

[BeU 86] describes a synthesis system which transforms a specification of the form

$$\{P(a)\}\ S\ \{x'=R(a)\}$$

into an imperative implementation of S. Output assertions are restricted to the form x'=R(a) in order to reduce the complexity of the problem. Apart from some transformations which deal with hierarchies of ADTs, there are essentially three transformation stages: specification transformation, data structure selection, and data structure and algorithm implementation. In the first stage, abstract constructs are transformed into algorithmic constructs. Transformations are applied to the abstract statement x:=R(a), (the solution for S based on the axiom of assignment); then, the transformed statement is decomposed into several more primitive assignments. Because the form of the output assertions is restricted, the transformations are rather weak; for example, more powerful transformations are contained in [HoH 86]. In the second stage, a cost analysis is performed and a set of data structures is chosen; the cost analysis and transformations are taken from [KaB 81]. The final stage integrates the chosen data structures with their algorithms and the algorithms are then transformed into PL\1.

1.4.3 Algebraic Specification

An attractive feature of algebraic specification is that implementation can be discussed within the same formal framework. Every programming language with a denotational semantics can be specified as an ADT. The problem of implementing an algebraic specification in a specific programming language can always be seen as a particular instance of the more general problem of implementing one ADT by another. There has been much research into this topic and it will be discussed further in chapter 5. Here, we mention some related work in the areas of direct implementation, transformation systems, efficiency analysis and automatic expert systems.

Specifications which fulfill certain requirements can be directly implemented by term rewriting or reduction systems. This kind of intuitive, but inefficient, implementation was suggested early on in [GHM 78b]; at present, there are several executable specification languages such as AFFIRM [Mus 80b] and OBJ [FGJ 85].

When the target language is specified as an ADT, then specifications can be *transformed* into specifications in the target language. The transformation rules can be verified by referring to the semantics of the specifications and to the notion of implementation. In most comprehensive transformation systems such as the CIP system [BBD 81], the transformations must be chosen manually. There are some less general systems in which the transformation strategy is automatic.

The strategy of [KaS 85] uses rewriting techniques to automatically implement (the initial algebra semantics of) one ADT by another. The user must specify both ADTs as confluent and terminating rewriting systems and the abstract relationship between the two ADTs must be given as a set of rewrite rules. An implementation cannot always be found using this strategy; this aspect will be discussed further in chapter 5.

[HsS 85] describes a similar approach within the logical programming framework. Here, PROLOG is an environment in which specification, implementation, and verification is carried out. ADTs are specified by defining the constructor operations as PROLOG "functors", and defined operations as predicates, or relations, on the constructor terms. Permutative axioms are allowed provided the relevant unification algorithm exists. Because the specifications are already executable, implementation is transformation into more efficient PROLOG code. Like [KaS 85], the user is required to supply both the implementing ADT and the implementations of the constructors of the implemented ADT.

The strategy proposed in [Moi 82] also uses rewriting techniques, but the user does not need to supply the implementing ADT nor the abstraction mapping. Instead, each specification is implemented by a generalised "Tree"-like ADT which is tailored to match certain aspects of the given ADT. For example, the arities of the "Tree" constructors depend, in a syntactic way, on the arities of the constructors of the given ADT.

In both [Moi 81] and [KaS 85], the efficiency of the implementation is not an issue. The implementations are just sets of recursion equations and there is no attempt to remove or reduce the levels of recursion. The literature contains very few references to the evaluation of the efficiency of representations of algebraic specifications; we briefly mention two rather different approaches to assigning cost measures to (the implementations of) operations. [CKS 87] considers direct implementation and presents an analysis of term algebras which is based on the cost of rewriting terms to their normal forms. [BoW 81] introduces the concept of a *performance* specification. A performance specification is derived from the algorithms, the representation, the particular machine for which the system is being designed, and the statistical properties of the actual data. The definition of implementation is not standard; instead, terms are represented by data flow graphs and the operations by control graphs. Each operation in the specification is assigned a cost equation which computes the operator costs over the whole system. The cost equations are derived from the data representation mapping, the algorithms which implement each operator by a control graph, and the probabilistic properties of operands associated with each operator.

To conclude this section, we mention two relevant programming expert systems.

The APE project, [EKR 80], [BOR 81], is most relevant because it appears to be the only attempt to automatically implement algebraically specified ADTs without using direct implementation or further information from the user. The implementations are constructed in INTERLISP using singly/doubly linked lists, association lists and array data structures. The approach is knowledge-based, and neither a summary of, nor justification, informal or formal, for the rules has been published. We summarise our understanding of this approach as follows. APE is an *Automatic Programming Expert* consisting of five phases; each phase is a production system containing the relevant "codified" knowledge. In the first phase, the *Functionality* phase, conclusions about the structure of the LISP implementations and the behaviour of the operations are drawn from an examination of the given signature. Each operation name has a *FUNCT* property associated with it. Eventually this property contains the implementing LISP function definition; after this phase it contains the function header: the function name, parameters

and result type. Operations are classified as having *reads*, *writes*, *deletes*, *singly-continuable*, or *double-continuable* behaviours. For example, assuming that T is the TOI, or type of interest, and D is the primitive type, then typical conclusions are:

- if σ is an operator with arity T -> D then σ is *reads* ,
- if σ is an operator with arity T,D -> T then σ is *singly-continuable,*
- if σ is an operator with arity T,T,D -> T then σ is *doubly-continuable.*

The second phase, the *Axiom* phase, analyses the given axioms in an attempt to infer further properties about the operations. For example, an operation may be classified as a *deletes* operation, or it may have the property: *works recursively through objects of type T.* Rules in this phase depend on the syntactic form of the equations and the properties inferred so far. To paraphrase, an example rule has the form:

- if the l.h.s. of an equation A has form X and the r.h.s. of A has form Y, and the outermost operation in X is σ, and σ is *singly-continuable/doubly-continuable,* then conclude that σ is *deletes/reads/writes.*

In the third phase, the *Representation* phase, the decisions about which data structure should be chosen for the implementation are made. For example, all ADTs which are constructed by *singly-continuable* operations are implemented by lists. Singly-linked, or doubly-linked, lists may be chosen and so there are rules to deduce whether an operation will read at the front or rear of the list; i.e. there are rules to conclude the property *reads(front)* or *reads(rear).* The rules in this phase depend mainly on the properties inferred during the previous phase. The fourth and fifth phases, the *Compilation* and *Lisp* phases resp., construct the actual LISP code and make optimisations and improvements. For example, the remaining non-LISP strings such as *reads(front)* in the *FUNCT* properties are expanded into LISP using a library of string to LISP expansions.

A knowledge-based approach is also described in [Per 86], as a contribution to the Esprit Meteor Project [Met 86]. An expert system to handle both automatic transformation and interactive transformation is being constructed; the work is still in progress. The operations are implemented by Ada packages using direct implementation techniques, and then the packages are optimised. The package declarations correspond to the operator arities, and the function bodies are defined by the relevant axioms. The code is optimised in two ways: by application of rule bases for special data type implementations, and by combining functions to avoid temporary data structures. Neither the content, nor the justification of the rule bases have yet been published and so we are unable to properly assess this approach.

1.5 Contribution of this Work

We conclude from an examination of the literature that the problem of automating the implementation of algebraically specified ADTs is a difficult one. In particular, the task of choosing representations for algebraic specifications is considerably more difficult than for constructive specifications.

In constructive specification, whilst efficient implementation is still difficult to achieve, representation choices can always be made because the domain of models is pre-defined; usually it is an assortment of sets, sequences, and tuples. Of course the standard representations may be inelegant or inefficient; much of the literature is concerned with the problem of choosing the best representations. On the other hand, in algebraic specification, the models are many-sorted algebras; the only straightforward representation choice is to implement Σ-terms by trees (i.e. by direct implementation).

The literature contains many discussions on the topic of how and when sets and sequences should be implemented, for example, by hash-tables or arrays, but there is little discussion of how and when the elements of an algebra should be implemented, for example, by hash-tables or arrays. One aim of this thesis is to define a framework in which representation choices can be discussed: the framework is provided by the *storage relations* and *storage graphs* derived from the initial Σ-algebra semantics of a specification.

We see, from the literature, that there are two approaches to constructing implementations: 1) by direct implementation as functional programs and then transformation/optimisation to imperative programs, or 2) by imperative data structures and implementation by imperative programs. We take the latter approach on the grounds that if imperative constructs are to be used, then they should be introduced as early as possible. If we must lose the attractive properties of functional programs, then we should at least gain some efficiency.

The APE work is interesting because it contains one of the few attempts to analyse an algebraic specification when choosing data structures. However, there are several criticisms to be made. The first is that the emphasis seems to be more on the coding and organisation of "programming knowledge" than on the nature and value of that knowledge. Even so, it appears from [Olt 85] that there are problems with the integrity of the system; there are conflicts between the rules and problems with persistence. We have three criticisms of the knowledge, or rules, contained in the system. The first is that the

behaviour classification such as *reads(front)*, *writes(oldest)*, *singly-continuable*, is too restrictive; certainly they are are common paradigms, but they are not exhaustive. Second, we would disagree with some specific rules. Because the rules appear to be based on examples and are not formally justified, it is difficult to see the reasoning behind some of the rules. Finally, the approach seems to be a syntactic one rather than a semantic one. For example, there is no mention of the resulting programs being provably correct. Moreover, the analysis depends on the syntactic form of the signature and equations, and not on the congruence generated by the equations. Unless this approach is based on some fundamental theorems relating the (semantic) congruence to the (syntactic) presentation, then the analysis is surely superficial.

We do not consider the knowledge-based approaches to be particularly successful for this problem; largely because without a semantic basis, the coding of large amounts of programming knowledge and experience is bound to lead to conflicts and inconsistencies. Programming experience has a part to play, for example, when deciding which properties of storage relations are important; however, we aim to abstract away from the textbooks such as [AHU 83] and [Mar 86], and the knowledge bases of APE [EKR 81], and discuss ***methodology***. We are concerned with *why*, *how*, and *when* decisions are taken, rather than *what* is the outcome.

Our approach bears some resemblance to the phases of APE: signature partitioning corresponds to the *FUNCT* phase, and *storage type* classifications correspond to the *AXIOM* phase. Unlike the APE literature however, we give formal definitions, based on the semantics of specifications, and motivate them, where possible, with discussion and examples.

We have retained the *spirit* of the digraphs used by D'Imperio and Rosenberg with our *storage relations* and *storage graphs*, but there are several important differences. The structure of our digraphs is unrestricted; moreover, each digraph is derived from an abstract data type, can be specified algebraically, and represents exactly one relation instead of several. Like [ShS 74], we distinguish a subset of the nodes of *storage relations*, but we make the distinction for reasons of efficiency. Like [Moi 82], we consider only one ADT: the *storage graph* ADT, as the implementing ADT. However, because the *storage relations* depend on the semantics of the given ADT, the *storage relations* are automatically tailored to the given ADT. Moreover, because the *storage relations* and *storage graphs* can be specified algebraically, we can enrich the implementing ADT with operations which increase the efficiency of the representation. In an algebraic framework, any desired operators can be synthesised by derived operators. In conclusion, we might say that another aim of this thesis is to resurrect the early

digraph approaches within the framework of algebraic specification and high-level implementation languages with user-defined data types.

Chapter Two
Keyless ADT Specifications and Operator Classifications

In this chapter we define some notation for the class of keyless, hierarchical ADT specifications. A partition is defined on the set of operators in the signatures of such specifications; this groups together operators with similar behaviour (for example, storing a data item or removing a data item). We show how operators are classified with respect to the partition and we discuss how an operator might be automatically classified.

2.1 Specifications

ADTs are specified using the order-sorted algebraic techniques developed by [FGJ 85], [GJM 85], [GoM 87a] and we use hierarchical constructs similar to those in [Bro 84]. We give a brief summary of the notation below: detailed definitions are contained in Appendix One.

An ADT ***specification*** consists of a ***regular order-sorted signature*** Σ and a set E of ***equations.*** When Σ is signature, we use S to refer its set of sort names; similarly for Σ' and S', Σ_b and S_b, Σ_0 and S_0, etc. $\leq$ refers to the partial order on the set of sorts. Operator symbols may be overloaded and for each operator σ, T_σ is the set of ***templates*** which specify the arities of σ. Given a signature Σ, a ***Σ-algebra*** is a many-sorted algebra whose carriers are indexed by the sorts S and whose operations correspond to the operators in Σ. In the following, when the distinction is obvious, we will confuse operators with operations. An ***order-sorted Σ–algebra*** generalises the many-sorted case; the sort ordering imposes a subset inclusion on the carriers. Regularity ensures that every term has a least sort and that overloaded operators are consistent. Given a regular order-sorted signature, T_Σ denotes the ***order-sorted term algebra*** of Σ. The congruence generated by E is denoted by $\equiv_E$. For a given congruence, [t] denotes the congruence class containing t. A ***(Σ,E)-algebra*** is an order-sorted Σ-algebra which satisfies the equations of E. A specification (Σ,E) determines a category of finitely-generated (Σ,E)-algebras; this has an initial object:

the quotient term algebra denoted by $T_{\Sigma,E}$. $T_{\Sigma,E}$ will be considered as the ***meaning*** of (Σ,E); we adopt initial algebra semantics so that proofs by induction are valid and that our definitions may be exploited using rewriting techniques.

Following [GuH 78], operators are divided into two classes: ***generators*** which generate all the values of the ADT and ***defined operators*** which are defined on those values. Terms which are constructed only from generators and variables are called ***generator terms***; terms which do not contain variables are called ***ground terms***. Generator terms are the obvious candidates for ***canonical***, or ***normal***, forms and we suggest that all specifiers should have such forms in mind when constructing a specification. Therefore, generators must be identified in the specification.

2.2 Specification Syntax

An ADT specification is presented in a concrete syntax which is a variant of the notation used in [EhM 85] and [FGJ 85]. A specification consists of five parts: the name of the specification is introduced by the keyword ***spec***, the sorts are introduced by the keyword ***sorts***, the partial order $\leq$ is introduced by the keyword ***subsorts***, the operator arities are introduced by the keyword ***gen*** for the generators and ***ops*** for the defined operators, the equations are introduced by the keyword ***eqns*** and the quantified variables and their sorts are given explicitly in each equation. Hidden operators are indicated in the signature by the keyword *(hidden)*; operators which are not hidden are called *visible*. Operators are assumed to be prefix unless they are indicated as infix by using the "_" symbol as a place holder. We illustrate the notation with an example; more examples are contained in Appendix Two.

```
spec       Nat_and_Pos =
sorts      nat, pos
subsorts   pos ≤ nat
gen        0      : nat
           succ   : nat -> pos
ops        _+_    : nat nat -> nat
           pred   : pos -> nat
eqns
∀x:nat.        x+0 = x
∀x:nat.        0+x = x
∀x,y:nat.      succ(x) + y = x + succ(y)
∀x:nat.        pred(succ(x)) = x
end
```

2.3 Hierarchical Specifications

We are interested in those ADTs which are built up, one by one, in a hierarchical way. The following definition is based on [Bro 84] and [WPP 83].

Definition: A specification (Σ,E) is called ***hierarchical*** iff Σ has a designated subsignature Σ_p (with sorts S_p) and E has a designated subset E_p consisting entirely of Σ_p-equations. The pair (Σ_p,E_p) is called the ***primitive*** specification. The primitive specification is designated by the ***basedon*** notation: a declaration of the form

```
spec H =
    basedon P
    sorts ...
    .....
```

designates the ADT **H** as the hierarchical specification ***over*** the primitive specification **P**.

The ***basedon*** construct is the only specification construct which may be used in the specification of the *object* ADTs (the ADTs which are going to be implemented). This construct is a simple one, but it is adequate for specification of the object ADTs; specifications of ADTs other than object ADTs will use the structuring constructs of CLEAR [BuG 81], [San 84].

2.3.1 Semantic Requirements

We want the hierarchy defined on the signature and equations to be clearly reflected in the algebra. Namely, the hierarchical specification should not impose any new equalities on the primitive sorts, nor generate any new terms of the primitive sorts.

These conditions are ensured by *hierarchical consistency* [BrW 82] and *sufficient-completeness* [GuH 78]. Hierarchical consistency demands that the congruence on the primitive sort generated by the primitive specification is unchanged; sufficient-completeness ensures that no new primitive values are generated: the operations of the primitive sorts should be totally specified and return only those values already defined in the primitive specification.

Definition: Let (Σ,E) be a hierarchical specification over (Σ_p,E_p). We say that (Σ,E) is ***hierarchically-consistent*** iff

$$(\forall \text{ sorts } s \text{ in } S_p)(\forall t,t' \in (T_\Sigma)_s)\ \ t \equiv_E t' \Leftrightarrow t \equiv_{Ep} t'.$$

Definition: Let (Σ,E) be a hierarchical specification over (Σ_p,E_p). We say that (Σ,E) is ***sufficiently-complete*** w.r.t. (Σ_p,E_p) (or a ***complete extension*** of (Σ_p,E_p)) iff every ground term of T_Σ of primitive sort is equivalent to a ground term of $T_{\Sigma p}$; i.e.

$$(\forall \text{ sorts } s \text{ in } S_p)(\forall t \in (T_\Sigma)_s)(\exists t' \in (T_{\Sigma p})_s)\ \ t \equiv_E t'.$$

Sufficient-completeness is undecidable in general, but may be proven in certain circumstances [GuH 78]. In the following we assume that specifications are hierarchical, hierarchically-consistent, and sufficiently-complete.

In order to prove the properties given in later sections, we will often require a form of completeness which is stronger than sufficient-completeness; namely, it will be necessary to ensure that there is at least one ground generator term in every congruence class induced by the equations. Completeness demands that the behaviour of *all* operations is totally specified; it is equivalent to proving a normal form lemma [GHM 78a] and is also, in general, undecidable. This property is guaranteed if the equations can be organised into a well-spanned, confluent and terminating rewriting system (modulo an equivalence relation with properties such as commutativity and associativity) [KaS 85].

The terminology of rewriting systems is contained in Appendix One. A terminating, confluent, rewriting system is a decision procedure for an equational theory

and can be derived, in certain circumstances, from a set of equations using the (possibly non-terminating) Knuth-Bendix completion procedure [KnB 70], [PeS 81].

2.3.2 Implementation and Choice of Semantics

The restriction to finitely-generated models is a very natural restriction; it ensures that all objects can be denoted by terms and hence proofs by induction are valid. The further restriction to initial semantics is perhaps not so natural; a loose semantics which is similar to the semantics given in [BMP 86] might be less restrictive. Clearly trivial terminal models are undesirable; by a *loose semantics* we mean any model which gives an initial interpretation to the primitive specification. Often, such a loose semantics is more intuitive from a programming point of view because the primitive sorts correspond to the input/output data types of programming languages and these are the only types that we can observe. The class of loose models is defined formally by:

> **Definition:** For a given hierarchical specification (Σ,E) over the primitive specification (Σ_p,E_p), we call the class of finitely-generated (Σ,E)-algebras ***Gen***(Σ,E). The class of all (Σ,E)-algebras A in ***Gen***(Σ,E) s.t. the Σ_p-reduct of A is an initial (Σ_p,E_p)-algebra is called ***Loose*** (Σ,E).

In general, we would like to choose any algebra from ***Loose*** (Σ,E) as the meaning for (Σ,E). However, initial models are desirable for provability: they exist, they are unique (up to isomorphism), and the associated term rewriting system provides a decision procedure for equality between ground terms. For these reasons, they are preferred. The initial algebra is the finest-grained algebra and so it is always a member of ***Loose*** (Σ,E); often it is the only member, in which case we call the specification *over-specified*. Many of the example specifications we consider are over-specified. A hierarchical specification which is not over-specified can become so by enrichment.

The implementations we shall construct will usually be implementations of the initial semantics of the object specification. In the cases where the object specification is not overspecified, we shall enrich the object specification to make it over-specified;

thus, in effect, we may implement a loose semantics (of the object specification).

2.4 Keyless and Keyed Specifications

Recall that a keyless ADT imposes a structure on the elements of the primitive sort which is *independent* of any prior relationship between the elements. The operations of storage and retrieval in these ADTs are specified by reference to some chronological ordering. On the other hand, a keyed ADT imposes a structure on the elements of the primitive sort which is *dependent* on some prior property of the elements. If the key is explicit, then the operations of storage and retrieval are specified by reference to some relation between indices and primitive elements. If the key is implicit, then they are specified by reference to some ordering relation between the primitive elements.

We can formalise the distinction between keyless and keyed ADTs by considering the form of specifications; our main concern is the specification of keyless ADTs and here we distinguish them by four conditions.

The first condition is a requirement for the primitive specification to contain a specification of **Bool** and one other sort with equality. In the terminology of CLEAR, this might be expressed by the following metatheory:

```
meta Prim =  enrich Bool by data sorts elem
end
```

(The ***data*** constraint always contributes an equality operator =, having arity s s -> bool, for each new sort s introduced.) The second condition restricts the hierarchical specification to having, in addition to the primitive sort, one new sort with a sub-sort (the former is usually referred to as the "type of interest" or "TOI"). The subsort is included to solve the constructor-selector problem [GoM 87b] and to ensure that the operations are total. The third condition excludes predicates other than equality so that retrieval by implicit key is excluded. The final condition restricts the arities of generators and primitive sort operators so that storage and retrieval by explicit key is excluded.

We now proceed to give the definitions of keyless and keyed specifications and at the same time we introduce some notation for referring to the sorts of an arbitrary

keyless or implicitly keyed specification.

Definition: A hierarchical specification (Σ,E) is designated as ***keyless*** when the following four conditions hold:

C1) The primitive specification includes the usual specification of **Bool** (see Appendix Two) and one other (discrete) sort. The latter sort is denoted by δ and is referred to as the ***primitive sort***. The primitive specification also includes an axiomatisation of equality on the primitive sort and equality is denoted by the operator `_=_:δδ -> bool`.

C2) Besides the primitive sorts, the specification contains exactly one other sort and its sub-sort. These sorts are denoted by τ and ϕ with $\phi \leq \tau$. We refer to τ as the ***derived sort*** and the subsort ϕ as the ***full_sort***. There are no constants of sort ϕ : i.e. $\Sigma_{\lambda,\phi}$ is empty.

C3) The specification contains no other operators (besides equality on the primitive sort), whose argument sorts include the primitive sort and whose result sort is `bool`.

C4) $(\forall w \in S^*)$ if $\Sigma_{w,\delta}$ is inhabited then $w \in \{\tau,\phi\}$,

$(\forall w \in S^*)(\forall s \in \{\phi,\tau\})$ if $\Sigma_{w,s}$ is inhabited then $w \in \{\tau,\phi,\delta\}^*$.

Definition: A hierarchical specification (Σ,E) is designated as ***implicitly keyed*** when conditions C1, C2, and C4 (above) hold but C3 does not hold.

Definition: A hierarchical specification (Σ,E) is designated as ***explicitly keyed*** when either conditions C1 and C2 (above) hold but C4 does not hold, or when C2 does not hold; i.e. C2 => (C1 $\wedge$ ~C4).

2.4.1 Examples

We illustrate these definitions with some examples. The following specifications, **Queue** and **Binary_Tree**, are keyless specifications but the specifications **Priority_Queue** and **Array** are implicitly keyed and explicitly keyed respectively. Appendix Two contains the primitive specifications **Nat**, **ANat**, **ONat** and **CNat**.

```
spec      Queue =
basedon   Nat
sorts     queue, neq
subsorts  neq ≤ queue
gen       eq       : queue
          add      : queue nat -> neq
ops       front    : neq -> nat
          dequeue  : queue  -> queue
eqns
∀d:nat.                 front(add(eq,d)) =  d
∀q:queue,d,d':nat.      front(add(add(q,d),d')) = front(add(q,d))
                        dequeue(eq) =  eq
∀d:nat.                 dequeue(add(eq,d)) =  eq
∀q:queue,d,d':nat.      .
dequeue(add(add(q,d),d'))=add(dequeue(add(q,d)),d')
end
```

```
spec          Binary_Tree =
basedon       ANat
sorts         tree, nonleaf
subsorts      nonleaf ≤ tree,
gen           m        : nat  -> tree
              comb     : tree tree -> nonleaf
ops           root     : tree -> nat
              left     : nonleaf -> tree
              right    : nonleaf -> tree
eqns
∀t,t':tree.   left(comb(t,t')) = t
∀t,t':tree.   right(comb(t,t')) = t'
∀t,t':tree.   root(comb(t,t')) = root(t) + root(t')
∀n:nat.       root(m(n)) = n
end
```

```
spec        Priority_Queue =
basedon     CNat
sorts       pq, fq
subsorts    fq ≤ pq
gen         eq          : pq
            add         : pq nat -> fq
ops         front       : fq -> nat
            dequeue     : fq  -> pq
            largest     : fq nat -> nat                    (hidden)
            last        : fq -> nat                        (hidden)
            rem         : fullq nat -> pq(hidden)
            cond        : bool pq pq->pq                   (hidden)
eqns
∀q:fq,d:nat.      front(q) = largest(q,last(q))
∀q:fq.            dequeue(q) = rem(q,largest(q,last(q)))
∀q:pq,d:nat.      last(add(q,d)) = d
∀d:nat.           largest(eq,d) = d
∀q:pq,d,d':nat    largest(add(add(q,d),d')) =
                        if d<d' then largest(q,d') else largest(q,d)
∀q:pq,d,d':nat.   rem(add(q,d),d') = cond((d=d'),q,add(rem(q,d'),d)
∀q,q':nat.        cond(T,q,q') = q
∀q,q':nat.        cond(F,q,q') = q'
end
```

```
spec          Array =
basedon       CNat
sorts         arr, frr
subsorts      frr ≤ arr
gen           nil         : arr
              assign      : arr nat nat ->frr
ops           lookup      : arr nat -> nat
eqns
∀a:arr,i,j,x:nat.
lookup(assign(a,i,x),j) =if (i==j) then x else lookup(a,j)
∀i,j,x,:nat.lookup(nil,j) = 0
end
```

We note that the inclusion of a subsort in this last explicitly-keyed ADT does not solve the problem of partiality of the selector: in order to totally specify the *lookup* operator, we must treat 0 as an 'underflow' position.

2.5 Operator Classification

In order to discuss arbitrary specifications, we impose a partitioning on signatures. The partitioning is based on the behaviour of the specified operations and it extends the usual classification of generators and defined operators. In [Tho 87], the operator class is given by the user; here, we give specification-independent definitions of the members of each partition class. The definitions are motivated by the need to analyse a specification in order to make implementation decisions and although the

definitions depend (implicitly) on a relation between terms, the relation is not a syntactic one. The classification bears some resemblance to the classification given in [BrW 81] and to the more recent classification of [Lan 87]; following our definitions, we contrast them with those given in [BrW 81] and [Lan 87].

The partition classes of a signature consist of the ***generator***, ***eliminator***, ***rearranging***, ***selector*** and any ***other*** operators. We denote the set of (given) generators of a signature Σ by Σ_g. Before we define the remaining partition classes, we explain the ideas behind the definition. Generators name the operations which define all the values of the specification. A selector names an operation which selects elements of the primitive sort from ground generator terms of the derived sort. An eliminator names an operation which eliminates, or removes, elements of the primitive sort from a ground generator term of the derived sort. For example, the tail operator `tl` is an eliminator in the usual specification of lists; the result of an application of `tl` to a list ground generator term (which is not equivalent to `nil`) is always another list which contains fewer terms of the primitive sort. A rearranging operator names an operation which preserves the primitive sort elements contained in a ground generator term of the derived sort. For example, the reverse operator `rev` is a rearranging operator in the usual specification of lists; the result of an application of `rev` to a list ground generator term is always another list containing exactly the same primitive terms, but the order of presentation has been permuted. The remaining operators are in the "others" partition. This class includes the hidden operators and any operators which appear to add primitive sort elements to derived sort elements or combine derived-sort elements, but which are not designated as generators; for example, the append operator `app` on lists (with the usual generators `nil` and `cons`) would be in this partition class.

Informally, the classification of an operator depends on the behaviour of the operation it names with respect to the multisets of primitive subterms of the (ground generator term equivalent of the) argument and result.

A function **leaves** is defined on ground generator terms; **leaves** returns the multiset of δ-sorted terms which occur in τ- and δ-sorted terms. In the following, we use the curly bracket notation, {} for set construction, and the double bracket notation, $[\![\,]\!]$, for multiset construction. The union operator, $\cup$, and the powerset operator, $\mathbb{P}$, will be overloaded:

Definition: Let (Σ,E) be a hierarchical specification with generators Σ_g.

leaves is the function from $T_{\Sigma g}$ to $\mathcal{P}((T_{\Sigma g})_\delta)$ defined by the equations:

leaves(d) $= [d]$ for $d:\delta$

leaves(c) $= []$ for $c \in \Sigma_{\lambda,\tau}$

leaves$(f(t_1,....,t_n)) =$ **leaves**$(t_1) \cup ... \cup$ **leaves**(t_n)

for $f \in \Sigma_{s1..sn,s}$,

$s \leq \tau$ and $t_1:s_1,..,t_n:s_n$.

The classes of selectors, eliminators and rearrangers are defined according to the behaviour of the operations with respect to **leaves. leaves** may not be well-defined on the congruence classes of $(T_{\Sigma g})_\tau$ (when E contains non-permutative equations between generator terms), therefore the definitions use terms and not classes. We note that although partially-ordered sorts simplify our approach to the handling of errors (i.e. we avoid them), some definitions must become more complex (because they involve specifications in general). When it is convenient to do so, we use a sort variable α to stand ambiguously for either or both of the sorts τ and ϕ.

Definition: A ***partitioned*** specification (Σ,E) is a keyless or implicitly keyed specification in which the operators are partitioned into 5 classes: $\Sigma_g, \Sigma_s, \Sigma_e, \Sigma_r, \Sigma_o$ (the generators, selectors, eliminators, rearrangers, and others resp).

The class of **generators**, Σ_g, is given in the specification and

$(\forall t \in T_\Sigma)\ (\exists t' \in T_{\Sigma g}: t' \equiv_E t)$.

A ***selector*** is any (visible) primitive sorted operator with domain sort α:

$\Sigma_s = \Sigma_{\phi,\delta} \cup \Sigma_{\tau,\delta}$.

A ***rearranger*** is a (visible) derived/full sorted operator which, when applied to a (ground generator) term, returns a (ground generator) term whose "leaves" are equivalent to those of the argument:

$$\Sigma_r = \{\sigma \mid (\forall \alpha \in \{\phi,\tau\}: \alpha \to \alpha \in T_\sigma)\ (\forall C \in (T_{\Sigma,E})_\alpha)(\forall t \in C : t \in (T_{\Sigma g}))$$
$$(\exists t' \in T_{\Sigma g}: t' \equiv_E \sigma(t)\ \wedge\ \mathbf{leaves}(t) = \mathbf{leaves}(t'))\ \}$$

An ***eliminator*** is an (visible) operator of the derived sort which is not a rearranger and when applied to a (ground generator) term, returns a (ground generator) term whose "leaves" are contained in those of the argument:

$$\Sigma_e = \{\ \sigma \mid \sigma \notin \Sigma_r \wedge$$
$$(\forall \alpha \in \{\phi,\tau\}: \alpha \to \tau \in T_\sigma)\ \ (\forall C \in (T_{\Sigma,E})_\alpha)(\forall t \in C : t \in (T_{\Sigma g}))$$
$$(\exists t' \in T_{\Sigma g}: t' \equiv_E \sigma(t)\ \wedge$$
$$(t \equiv_E t'\ \wedge\ \mathbf{leaves}(t') = \mathbf{leaves}(t))$$
$$\vee\ (\mathbf{leaves}(t') \subset \mathbf{leaves}(t)))))\}$$

Operators are also classified in [BrW 81] and [Lan 87]; the following two subsections describe those classifications and compare them with the above classification.

2.5.1 Comparison with the Broy-Wirsing Classification

[BrW 81] contains a classification of operations as *constructor*, *destructor*, *output* and *rearrangement* operations. In this section we give the definitions relevant to this classification and contrast them with the ones given above. The definitions actually given in [BrW 81] involve specifications with loose, partial-algebra semantics; we have adapted these definitions to the restricted case of initial (total) algebra semantics.

Definition: [BrW 81] Let (Σ,E) be a specification with generators Σ_c. The ***subterm*** ordering $<$ is the least transitive relation such that for $f: s_1 \dots s_n \rightarrow s$ and all terms $t_1:s_1, \dots, t_n:s_n$ in T_Σ,

$$(\forall i:1\leq i\leq n)\ t_i < f(t_1,\dots,t_n).$$

$\leq$ denotes the reflexive closure of $<$.

The transitive relation $<_{\Sigma c}$ is defined by:

$t_1 <_{\Sigma c} t_2$ iff

$$(\forall t \in T_{\Sigma c})\ t \equiv_E t_1 \Rightarrow (\exists t' \in T_{\Sigma c})\ t < t' \wedge t' \equiv_E t_2$$

$$\wedge \quad (\forall t' \in T_{\Sigma c})\ t' \equiv_E t_2 \Rightarrow (\exists t \in T_{\Sigma c})\ t < t' \wedge t_1 \equiv_E t.$$

$\leq_{\Sigma c}$ denotes the reflexive closure of $<_{\Sigma c}$.

An operation $f: s_1 \dots s_n \rightarrow s$ is called

- a ***(pure) constructor*** iff for all terms $t_1:s_1, \dots, t_n:s_n$ in T_Σ, when for each i, $1\leq i\leq n$, t_i is of sort s then we have

$$t_i <_{\Sigma c} f(t_1,..,t_n),$$

- a ***(pure) destructor*** iff for some i, $1\leq i\leq n$, $s_i=s$ and for all terms $t_1:s_1, \dots, t_n:s_n$ in T_Σ we have

$$f(t_1,..,t_n) <_{\Sigma c} t_i,$$

- a ***rearrangement*** iff for all terms $t_1:s_1, \dots, t_n:s_n$ in T_Σ, when for each i, $1\leq i\leq n$, such that $s_i=s$, we have

$$(t_i \leq_{\Sigma c} f(t_1,..,t_n) \vee f(t_1,..,t_n) \leq_{\Sigma c} t_i) \Rightarrow (t_i \equiv_E f(t_1,..,t_n)).$$

- an ***output*** function if the specification is hierarchical, s is a primitive sort and at least one of the $s_1 \dots s_n$ is a hierarchical sort.

Hereafter, this classification is referred to as the *BrW* classification.

Whereas our classification is based on the behaviour of functions with respect to the relationships between the *multisets* of primitive subterms in the argument and the result, the *BrW* classification is based on the behaviour of functions with respect to the subterm relationship between the argument and the result. It is easy to see that when there are no equations between generator terms, then a generator is a *BrW* (pure) constructor. If the generators are associative, or commutative, or both, then the generators are also (pure) constructors. Only "contrived" equations between generators, for example, equating two generator terms such as `push(create,0)` and `create`, or making a generator commutative for only one argument such as the equation

`add(add(emptyset,0),succ(0))=add(add(emptyset,succ(0)),0),`

will prevent generators from being (pure) constructors.

A rearranging operation is always a (*BrW*) rearrangement operation, but not the converse. A selector operation is always a (*BrW*) output operation, but not all eliminators are (*BrW*) destructors. Clearly operations which are inverses to constructors such as *pop* in **Stack** and *left* and *right* in **Binary_Tree** are destructors, but *dequeue* in **Queue** is not a destructor; i.e. we cannot show that

$$\texttt{dequeue(add(add(eq,0),succ(0)))} <_{\Sigma_C} \texttt{add(add(eq,0),succ(0))}$$

where $\Sigma_C = \{\texttt{eq}, \texttt{add}\}$. Under the *BrW* classification, *dequeue* would be a rearrangement operation; i.e. for all $\texttt{t} \in (T_\Sigma)_{\texttt{neq}}$ we can show that `t` and `dequeue(t)` are incomparable in $\leq_{\Sigma C}$ and that

$$\texttt{dequeue(eq)} \leq_{\Sigma C} \texttt{eq} \Rightarrow \texttt{eq} \equiv_E \texttt{dequeue(eq)} .$$

So, *dequeue* is (trivially) a rearrangement operation.

Although the *BrW* classification was defined with a different motivation in mind: the definition of behaviour term algebras and the classification of terms by contexts, it is not obvious that the restriction of destructors to the (syntactic) inverses of constructors is necessary or even desirable. Certainly for our purposes a definition based on the *inclusion* of multisets of the primitive sorted terms occurring in ground generator terms is more appropriate.

2.5.2 Comparison with the Lange Classification

[Lan 87] contains a classification of operations as *constructor*, *selector*, and *test* operators. The motivation for this classification is to give an algebraic framework for applying inductive inference methods to program synthesis. We shall consider this topic again in chapter 5; in this section we give the definitions of [Lan 87] using our notation and contrast them with the ones given above. In the following definitions, X is a set of sorted variables containing x and y, and [t/x] denotes the substitution of term t for the variable x.

Definition: [Lan 87] A ***classified*** specification (Σ,E) is partitioned into 3 classes: Σ_c, Σ_{se}, and Σ_t (the constructors, selectors, and test operators resp.) such that the following three conditions hold:

i) $(\forall t \in T_\Sigma)\ (\forall t \notin (T_\Sigma)_{bool})\ (\exists t' \in T_{\Sigma c})\ t' \equiv_E t.$

ii) $(\forall t \in T_{\Sigma c})\ (\exists k \in N:\ \exists t_1,\ldots,\exists t_k \in T_{\Sigma c})\ (\forall s \in T_{\Sigma se}(\{x\}))$

$$s[t/x] \equiv_E t_1 \vee \ldots \vee s[t/x] \equiv_E t_k.$$

iii) $(\forall f \in T_{\Sigma t})\ (\forall t \in T_{\Sigma c})\ (\forall s \in T_{\Sigma c}(\{x\}))$

$$f(s[t/x]) \equiv_E T \Rightarrow (\forall s' \in T_{\Sigma c}(\{y\}))\ f(s'[s[t/x]/y])) \equiv_E T.$$

(n.b. T is the boolean constant)

Hereafter this classification is referred to as the *Lange* classification.

In addition to the operator classification, a notion of *size* measure for the normal forms of terms is required. A size measure is a function from the normal forms of T_Σ to N and has the properties that there are only finitely many normal forms with a particular size, constructors increase the size of normal forms, and selectors decrease the size of normal forms.

Definition: [Lan 87]

Let (Σ,E) be a classified specification which is a confluent, terminating term rewriting system (when the equations are considered as left to right rewrite rules) with normal forms $T_{NF(\Sigma)}$ and let NF(t) be the normal form of t. Let # be a function #: $T_{NF(\Sigma)}$ -> N, # is a ***compatible size measure*** iff

iv) $(\forall n \in N)$ $\#^{-1}(n)$ is finite.

v) For every non-constant operator f: $s_1 \ldots s_n \rightarrow s \in T_{\Sigma c}$, for terms $t_1 \in T_{\Sigma c}, \ldots, t_n \in T_{\Sigma c}$, for i=1,...,n,

$$\#NF(t_i) \leq \#NF(f(t_1,\ldots,t_n)).$$

vi) $(\forall s \in T_{\Sigma se}(\{y\}))$ $(\forall t \in T_{\Sigma c})$

$$(\#NF(s[t/y]) < \#NF(t)) \vee s[t/y] \equiv_E t.$$

There are several similarities between our operator classification and the **leaves** function and the *Lange* classification and the compatible size measure; we explain the similarities in the following.

From i) we can easily see that our generators are (*Lange*) constructors, and from iii) we can see that any `bool`-sorted operator is a test operator. Condition ii) says that for every constructor term, there is no infinite chain of (*Lange*) selectors which can be applied. The set of eliminator and selector operators forms a set of (*Lange*) selectors because selectors can only be applied once and eliminators strictly decrease the "leaves" of a term: there can be no infinite chains. Namely, by the definition of Σ_e, we can show that for all $t:\tau$ the set $\{[e^n(t)] | e \in \Sigma_e, n \in N\}$ is finite and hence the set $\{[s(E(t)]] \; s \in \Sigma_s, E \in (\Sigma_e)^*\}$ is finite. (Because of the arity of the selectors we need not consider $(\Sigma_e \cup \Sigma_s)^*$).

A rearranger is not necessarily a (*Lange*) selector; in the following, we will call a rearranger $\sigma : \alpha \rightarrow \alpha$, where $\alpha \in \{\phi,\tau\}$, ***stable*** when σ has the property

$$(\forall t{:}\alpha)\ (\exists n \in N)\ \sigma^n(t) =_E t.$$

A stable rearranger is also a (*Lange*) selector. For example, the `rev` operator is a stable rearranger in the **`Reversible_list_1`** specification and the `shift` operator is a stable rearranger in the **`Circular_List_right`** specification. In the former, n is always 2; i.e. $(\forall t : \texttt{list})\ \texttt{rev}(\texttt{rev}(t)) \equiv t$ is in the inductive theory of E. In the latter example, n is proportional to the size of the (normal form) of the argument. We call a stable rearranger σ an ***n-stable*** rearranger, for some $n \in N$, when

$$(\forall t{:}\alpha)\ \sigma^n(t) =_E t.$$

For example, `rev` is a 2-stable rearranger.

Now, we consider **leaves** as a size measure. Of course **leaves** has the wrong arity to be a size measure, but **leaves** is easily composed with a multiset encoding function (using the unique factorisation theorem) to form a compatible size measure. To encode multisets of sort X, we first give an injection from (the carrier of) X to the prime numbers, then we encode each multiset by the product of the prime number image of each element of X raised to the power of the number of occurrences of that element in the multiset. Taking the ground generator terms as normal forms, we can then consider the generators as (*Lange*) constructors, eliminators and 1-stable rearrangers as (*Lange*) selectors.

Definition: Let (Σ,E) be a keyless specification, let $\rho : (T_{\Sigma g})_\delta \rightarrow N$ be an injection which maps primitive ground generator terms to prime numbers, let mset be the set of multisets of primitive ground generator terms and $o{:}\text{mset} \times (T_{\Sigma g})_\delta \rightarrow N$ be the function which maps a multiset S and a primitive ground generator term d into the number of occurrences of d in S.

We define ***code*** : mset -> N to be the function which encodes multisets in N by

$(\forall S \in \text{mset})$ **code**(S) $=_{def} \prod_{d:D} \rho(d)^{o(S,d)}$

where $D =_{def} (T_{\Sigma g})_{\delta}$ and $\prod_{d:D}$ is the arithmetical product indexed by terms from D.

Now we can describe the circumstances under which the encoding of **leaves** is a compatible size measure for our operator classification. Essentially, the circumstances are straightforward; the only complication arises from the fact that generators with arity τ -> τ could cause property iv): the inverse image of everything in the codomain of the size measure is finite, to be violated. Thus, such operators must be excluded.

Proposition: Let (Σ,E) be a keyless specification which is a confluent, terminating term rewriting system with ground generator terms as normal forms and fulfills the condition that
either
a) all non-constant generators include the primitive sort in their domains,
or
b) there are no constant generators of the derived sort and generators of the derived sort which do not include the primitive sort have arity > 1.

When we define Σ_c as Σ_g and Σ_{se} as $(\Sigma_e \cup \Sigma_{r'})$, where $\Sigma_{r'}$ is the set of 1-stable rearrangers, then **code**∘**leaves** is a compatible size measure.

proof: **code** provides a unique representation in N of each multiset in the codomain of **leaves** and in addition

S1) **leaves**(t) $\subset$ **leaves**(s) => **code**∘**leaves**(t) < **code**∘**leaves**(s).

code∘**leaves** satisfies the conditions of a compatible size measure in the following way:

iv) There are only a finite number of ground generator terms with a particular size measure. Recall that **leaves**(t) consists of the primitive subterms of t and that generators are either constants or contain δ in their domain. Conditions a) and b) in the Proposition ensure that the size of **leaves**(t) increases with each application of a derived-sort generator; thus for any multiset of primitive normal forms $\{x_1,...,x_n\}$, the number of terms which can be constructed from $\{x_1,...,x_n\}$ and the τ-sorted generators, using the members of $\{x_1,...,x_n\}$ exactly once, is finite. **code** is an injection and so $(\forall n \in N)$ $(\mathbf{code} \circ \mathbf{leaves})^{-1}(n)$ is finite.

v) Constructors increase the size of ground generator terms. By the definition of **leaves**, for any generator f: s_1 ... s_n -> s, (n>0),

$(\forall t_1:s_1, \ ... \ ,t_n:s_n)(\forall i:1 \leq i \leq n)$

$\mathbf{leaves}(f(t_1,...,t_n)) \subseteq \mathbf{leaves}(t_i)$.

Thus, by S1:

$(\forall t_1:s_1, \ ... \ ,t_n:s_n)$ $(\forall i:1 \leq i \leq n)$

$\mathbf{code} \circ \mathbf{leaves}(f(t_1,...,t_n)) \leq \mathbf{code} \circ \mathbf{leaves}(t_i)$.

vi) Selectors strictly decrease the size of ground generator terms. By definition of eliminators, for any eliminator f: α -> τ, $\alpha \in \{\phi,\tau\}$,

$(\forall t \in (T_{\Sigma g})_\alpha)(\exists t' \in (T_{\Sigma g})_\alpha : f(t) \equiv_E t')$ $(t \equiv_E t')$ $\vee$ $(\mathbf{leaves}(t') \subset \mathbf{leaves}(t))$.

By the definition of 1-stable rearrangers, for any 1-stable rearranger $f:\alpha \to \alpha$, $\alpha \in \{\phi,\tau\}$,

$(\forall t \in (T_{\Sigma g})_\alpha)(\exists t' \in (T_{\Sigma g})_\alpha : f(t) \equiv_E t')$ $(t \equiv_E t')$.

Thus, by S1, for all eliminators and 1-stable rearrangers f:

$(\forall t \in (T_{\Sigma g})_\alpha)$ $(\exists t' \in (T_{\Sigma g})_\alpha : f(t) \equiv_E t')$

$t \equiv_E t' \ \vee \ (\mathbf{code} \circ \mathbf{leaves}(t') < \mathbf{code} \circ \mathbf{leaves}(t))$ $\square$.

Some examples of keyless specifications which fulfill the conditions of the above proposition are **Stack**, **SBinary_Tree**, **List_app**, **Binary_Tree**, **Reversible_List_1**, **Stack2**, **Queue**, **List**, **Binary_Tree_2**,

Circular_List_right, and **Circular_List_left**. We note that in each of these specifications only the eliminators are (*Lange*) selectors as there are no 1-stable rearrangers.

When considering **code∘leaves** as a compatible size measure, neither selectors nor n-stable, n>1, (or even just stable) rearrangers are included as (*Lange*) selectors. Selectors are not included because although they meet condition iii), they do not necessarily meet condition vi). For example, in the **Stack** specification, *top* does not always decrease the size of the "leaves" of terms; for example,

code∘leaves(top(push(create,0))) < **code∘leaves**(0)

is not true, and

top(push(create,0)) $\equiv_E$ push(create,0)

is ill-typed. Stable rearrangers are not included because rearrangers preserve the size measure of a term and they typically become stable after more than one application, thus condition vi) does not hold; if the < relation in condition vi) is relaxed to a $\leq$ relation, then condition vi) would hold for stable rearrangers.

Alternatively, we could add an additional class to this classification; unlike the *BrW* classification, the *Lange* classification does not include any notion of rearranging, or permuting operations. We would suggest that an additional class of rearrangers Σ_{re} could be defined as the class containing those operations which satisfy condition iii) and the following condition vii):

vii) $(\forall s \in T_{\Sigma re}(\{y\}))\ (\forall t \in T_{\Sigma c})$ #NF(s[t/y]) =#NF(t).

Thus, any stable rearranger would be in Σ_{re}; for example, *rev*, *shift* and *switch*, from **Reversible_List_1**, **Circular_List_right**, and **SBinary_Tree** resp., would all be members of the resp. Σ_{re}.

We note that although the stability of a rearranger is not required for our approach (c.f. chapter 3), it is a reasonable requirement and it may be possible, in the future, to combine these two classifications.

2.6 Classifying Operators

The classification of an operator (as a selector, eliminator, etc.) is easily determined when the specification can be organised into a terminating, confluent set of rewrite rules. Consider the following examples.

Example 2.1: The *dequeue* operator in the **Queue** specification is an eliminator.

Proof: Let R1 to R7 be the rewrite rules for **Queue** given in Appendix Two. We have checked that they form a terminating, confluent set of rules using the Knuth-Bendix completion procedure in ERIL [Dic 87]. In this particular example, there are no equivalences between generator terms and so **leaves** is well-defined; the normal forms are just the generator terms.

We will use the notation $add^n(eq, x_1 \ldots x_n)$ to abbreviate the term $add(add(\ldots(add(eq, x_1), \ldots), x_{n-1}), x_n)$, where $x_1, \ldots, x_n$ are variables of sort *nat*.

First, we require a simple lemma which we prove by induction.

lemma: $dequeue(add^{n+2}(eq, x_1 \ldots x_{n+2})) \Rightarrow add^{n+1}(eq, x_2 \ldots x_{n+2})$

proof:

Base case: (n=0)

$dequeue(add^2(eq, x_1, x_2)) \Rightarrow add(dequeue(add(eq, x_1), x_2))$ (R5)

$\Rightarrow add(eq, x_2)$ (R4)

Induction step: Assume $n \in N$ and

$dequeue(add^{n+2}(eq, x_1 \ldots x_{n+2})) \Rightarrow add^{n+1}(eq, x_2 \ldots x_{n+2})$.

Consider $dequeue(add^{n+3}(eq, x_1 \ldots x_{n+3}))$.

$dequeue(add^{n+3}(eq, x_1 \ldots x_{n+3}))$

$\Rightarrow add(dequeue(add^{n+2}(eq, x_1 \ldots x_{n+2})), x_{n+3})$ (R5)

$\Rightarrow add(add^{n+1}(eq, x_2 \ldots x_{n+2}), x_{n+3})$ (ass.)

$\Rightarrow add^{n+2}(eq, x_2 \ldots x_{n+3})$ (abbr.)

Conclusion: $dequeue(add^{n+2}(eq, x_1 \ldots x_{n+2})) \Rightarrow add^{n+1}(eq, x_2 \ldots x_{n+2})$

for all n. □

Second, we show that for all normal forms $t : queue$, there is another normal form t' which is the normal form of $dequeue(t)$ and

$$(t =_E t' \wedge \textbf{leaves}(t') = \textbf{leaves}(t))$$

$$\vee\ (\textbf{leaves}(t') \subset \textbf{leaves}(t)).$$

We suppose that t has the form $add^m(eq, t_1 \ldots t_m)$ and examine three cases:

Case 1: (m>1)

let t' be the term $add^{m-1}(eq, t_2 \ldots t_m)$.

$dequeue(add^m(eq, t_1 \ldots t_m)) \Rightarrow t'$ (lemma 1)

$\textbf{leaves}(t') = [\![t_2, \ldots, t_m]\!]$ (defn,induction)

$\textbf{leaves}(add^m(eq, t_1 \ldots t_m)) = [\![t_1, \ldots, t_m]\!]$ (defn,induction)

i.e. $\textbf{leaves}(t') \subset \textbf{leaves}(add^m(eq, x_1 \ldots x_m))$

Case 2: (m=1)

let t' be the term eq.

$dequeue(eq) \Rightarrow t'$ (R4)

$\textbf{leaves}(t') = [\!]$ (defn)

$\textbf{leaves}(add(eq, t_1)) = [\![t_1]\!]$ (defn,induction)

i.e. $\textbf{leaves}(t') \subset \textbf{leaves}(add(eq, t_1))$

Case 3: (m=0)

let t' be the term eq; $t = t' = eq$.

$dequeue(eq) \Rightarrow t'$ (R4)

i.e. $(eq =_E t') \wedge (\textbf{leaves}(t') = \textbf{leaves}(eq)$. □

Example 2.2: The *shift* operator in the **Circular_List_right** specification is a rearranger.

Proof: Let R1 to R7 be the rewrite rules for **Circular_List_right** given in Appendix Two. We have checked that they form a terminating, confluent set of rules using the Knuth-Bendix completion procedure in ERIL [Dic 87]. In this example, there are also no equivalences between generator terms and so **leaves** is well-defined; the normal

forms are just the generator terms.

We will use the notation $x_{n+1}:\ldots:x_1$ to abbreviate the term $x_{n+1}:(\ldots(x_1:nil),\ldots)$, where $x_1,\ldots,x_n$ are variables of sort *nat*.

First, we require a simple lemma which we prove by induction.

lemma: $shift(x_{n+1}:\ldots:x_1) \Rightarrow x_n:\ldots:x_1:x_{n+1}$.

proof:

Base case: (n=0)

$shift(x_1) \Rightarrow x_1$. (R4)

Induction step: Assume $n \in N$ and

$shift(x_{n+1}:\ldots:x_1) \Rightarrow x_n:\ldots:x_1:x_{n+1}$.

Consider $shift(x_{n+2}:\ldots:x_1)$.

$shift(x_{n+2}:\ldots:x_1) \Rightarrow x_{n+1}:shift(x_{n+2}:x_n\ldots:x_1)$ (R5)

$\Rightarrow x_{n+1}:x_n\ldots:x_1:x_{n+2}$ (ass.)

Conclusion: $shift(x_{n+1}:\ldots:x_1) \Rightarrow x_n:\ldots:x_1:x_{n+1}$ for all n. □

Second, we show that for all normal forms $t:nel$, there is a normal form t' which is the normal form of $shift(t)$ and **leaves**(t') = **leaves**(t).

We suppose that t has the form $t_{n+1}:\ldots:t_1$ and let t' be the term $t_n:\ldots:t_1:t_{n+1}$.

$shift(t_{n+1}:\ldots:t_1) \Rightarrow t'$ (lemma 1)

leaves$(t') = [\![t_{n+1},\ldots,t_1]\!]$ (defn,induction)

leaves$(t_{n+1}:\ldots:t_1) = [\![t_{n+1},\ldots,t_1]\!]$ (defn,induction)

i.e. **leaves**(t') = **leaves**(t). □

2.6.1 Automating the Operator Classification

We can see from the definitions that the classification of an operator will always require a proof by induction; this immediately suggests that the rewriting theorem proving technique called *inductionless induction* or *proof by consistency* [Mus 80a]

[HuH 82], [GoM 83], [KaM 87] might be appropriate.

2.6.1.1 Rewriting Techniques

Inductionless induction methods aim to decide whether, given two terms M and N in $T_\Sigma(X)$ (terms constructed from the operator symbols and a countably infinite set of variables X) the equation M = N is an inductive theorem of a given rewriting system R. The methods are based on the Knuth-Bendix (KB) completion procedure (or variations thereof) and, essentially, if the algorithm completes on R U {M = N}, then M = N is an equation in the inductive theory. An equation in the inductive theory is not necessarily valid in all models; however, this method is appropriate when using initial algebra semantics.

The approach of [Mus 80a] requires equality predicates and thus an axiomatisation of Bool; if the algorithm completes on R U {M = N} then the resulting theory must be consistent (i.e. true $\equiv_E$ false must not be derivable). The approach of [HuH 82] does not require an equality predicate; instead, the completion algorithm is extended. In order to apply the (extended) algorithm, the specification must satisfy the *Principle of Definition*. A specification (Σ,E) with generators Σ_g has the *Principle of Definition* when every ground term is equivalent to a unique ground generator term; i.e.

i) $(\forall t \in T_\Sigma)(\exists s \in T_{\Sigma g})\ t \equiv_E s$

ii) $(\forall s,t \in T_{\Sigma g})\ s \equiv_E t$ implies s=t.

We have tried, without success, to classify operators automatically using the rewrite rule laboratory Reve 2.4 [FoG 84] (using the approach of [HuH 82]) and with ERIL [Dic 87] (using the approach of [Mus 80a]).

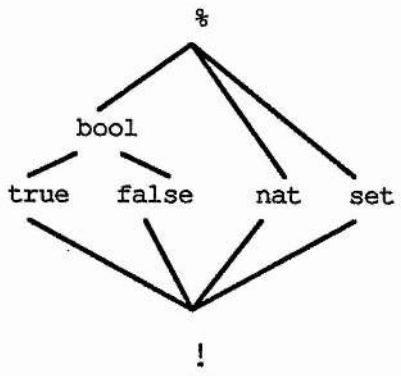
Problems arise because before an operator can be classified, the axioms for **leaves** and (finite) multisets of primitive sort terms must be added to each specification. The enriched specification must then be organised into a terminating and confluent rewriting system. There are no difficulties with the axioms for **leaves**; for example, in the **Queue** specification, we would add an additional sort, `set` say, with generators `_|_: nat set -> set` and `[]: set`, the operator `leaves: queue -> set`, and the following equations:

```
∀n:nat,q:queue.  leaves(add(q,n)) = n | leaves(q)
                 leaves(eq)  = []
```

The usual specification of multisets has an associative-commutative generator. Although Reve allows associative-commutative operators, all generators with these properties violate the *Principle of Definition* and so the classifications cannot be checked within the current version of Reve. The *Principle of Definition* is further extended in [Mit 87] to allow for alternate sets of constructors but this does not solve our problems. However, there are indications in [HuH 82] that the method could be extended to handle associative-commutative generators and [JoK 86] contains some algorithms, although we are not aware of any implementations of these results at this time.

ERIL (Equational Reasoning Interactive Laboratory) includes an extension to the Knuth-Bendix completion algorithm which allows for the treatment of order-sorted algebras. By using a suitable axiomatisation of Bool, we can perform inductive proofs in ERIL using the inductionless induction approach of [Mus 80a]. When we use the axiomatisation of Horn Clauses given in [Pau 85], then we also have a form of positive conditional equations. ERIL does not include associative-commutative unification and so it cannot allow the associative-commutative nature of multiset construction. As an alternative, when there is a total order on the primitive terms, then we can give a specification in which each multiset has a normal form which is an ordered list. Such a specification is really more of an *implementation* of multisets (c.f. chapter five). This approach is also similar to an approach proposed in [Tho 86] where "laws" (rewrite rules) are introduced into ADTs. The types with laws can be thought of as sub-types of the associated free type and the elements of a sub-type are the elements in normal form.

We give below an (ERIL) example specification of multisets and ordered lists of natural numbers. To aid the reader, we also give a lattice of the sorts. Comments are given in between "/*" and "*/"; and when the right hand side of a bool sorted equation is T, then it is omitted. The actual ERIL listings for this specification (and the examples from Appendix 2) are given in Appendix 4.



lattice of sorts

sorts !, %, true, false, bool, nat, set

subsorts ! ≤ %, ! ≤ true, ! ≤ false, ! ≤ bool, ! ≤ nat, ! ≤ set,
true ≤ %, false ≤ %, bool ≤ %, nat ≤ %, set ≤ %,
true ≤ bool, false ≤ bool

ops

```
T       : true
F       : false
_v_     : bool bool -> bool              /*bool ops*/
_^_     : bool bool -> bool
_=>_    : bool bool -> bool
~       : bool -> bool
_=_     : % % -> bool
_=_     : % ! -> bool
_=_     : ! % -> bool
_=_     : nat nat -> bool
_=_     : set set -> bool
_=_     : bool bool-> bool
_=_     : true false->false
_=_     : false true->false
0       : nat
succ    : nat -> nat                     /*nat ops*/
_<_     : nat nat ->bool
[_]     : nat -> set                     /*set ops*/
_|_     : nat set -> set                 /*unordered insert*/
[_|_]   : nat set -> set                 /*ordered insert*/
[]      : set
_⊆_     : set set -> bool                /*inclusion*/
_mem_   : nat set -> bool                /*membership*/
```

```
eqns ∀n,n':nat, b,b' : bool, S,S' : set.

~T = F
~F = T
T ∨ b = T
b ∨ T = T
b => b' = ~b ∨ b'
~(b ∧ b') = ~b ∨ ~b'
b = b                          /*reflexivity of =*/
n = n                          /*must be specified*/
S = S
n < 0 = F
n  < n = F
0 < succ(n)
succ(n) < succ(n') = n < n'

(0 = succ(n)) = F
(succ(n) = 0) = F
(succ(n) = succ(n')) = (n = n')

n | [] = [n]
[n | []] = [n]
(n = n') => n | [n' | S] = [n' | S]
(n < n') => n | [n' | S] = [n  |[n' | S]]
(n < n') => n' | [n | S] = [n | [n' | S]]

n mem [] = F
(n = n') => n mem [n' | S]
(n < n') => (n mem [n' | S] = F)
(n < n') => (n' mem [n|S] = n' mem S)

[] ⊆ S
[n | S] ⊆ S' = (n mem S') ∧ (S ⊆ S')
```

As an example, if we were to enrich the **Queue** specification with the above rules and the axiomatisation of **leaves**, then in order to classify *dequeue* as an eliminator using inductionless induction, we would add the following rule:

(T) `leaves(dequeue(add(q,n))) ⊆ leaves(add(q,n)).`

Unfortunately, running the Knuth-Bendix completion algorithm in ERIL (using the inductive ordering) on the above equations produces an infinite set of rewrite rules. Of course a finite portion of an infinite set of rules may be used as a semi-decision procedure and thus some interesting theorems may be derivable in finite time, but this approach is not adequate for our purposes. For example, the rule (T) is not derivable (in finite time) from the above system.

Infinite sets of rewrite rules may be avoided either by altering the completion procedure (and possibly weakening the results), or by enriching the original set of rewrite rules.

The completion algorithm given in [Fri 86] is often successful where the KB completion algorithm loops because confluence is, in general, too strong for inductive proofs. [Fri 86] shows that ground confluence (confluency for ground terms) along with a certain property of critical pairs (called complete superposability) is sufficient to guarantee the validity of a theorem in the initial algebra. [Gan 87] also contains a completion algorithm based on ground confluence which gives a weaker result than standard KB completion; coincidentally, this paper uses the specification of ordered lists as an example of a specification for which the algorithm terminates.

In [JaT 87], inductive inference techniques are used to synthesise an enrichment of a rewriting system such that the KB completion algorithm terminates when it is applied to the enriched set of rules. The inductive inference techniques [Bar 83] can synthesise new operators and new rewrite rules; in this case a new operator and some rewrite rules expressing the weakest common generalisation of the (previously) infinite set of critical pairs are synthesised. The example used is the rewriting system which consists of (part of) the **Queue** specification, **(R1)**,...,**(R3)**, a specification of leaves, **(R4)** and **(R5)**, a very minimal specification of 'sets', **(R6)** and **(R7)**, and **(T)**. The signature and variable quantification are omitted; -> denotes the rewriting relation.

(R1) dequeue(eq) -> eq
(R2) dequeue(add(eq,d)) -> eq
(R3) dequeue(add(add(q,d),d1)) -> add(dequeue(add(q,d)),d1)
(R4) leaves(add(q,d)) -> d | leaves(q)
(R5) leaves(eq) -> []
(R6) [] $\subseteq$ S -> T
(R7) (d | S) $\subseteq$ [] -> F
(T) leaves(add(q,d)) $\subseteq$ leaves(dequeue(add(q,d))) -> T

The rewriting system consisting of only **(R1)**,...,**(R7)** is terminating and confluent; the addition of **(T)** results in an infinite set of rules. This is not a surprising result as the theory of 'sets'; i.e. **(R4)**, **(R5)**, **(R6)** and **(R7)**, is very inexpressive. Indeed, **(R4)**, **(R5)**, **(R6)** and **(R7)**, together form a specification *requirement* for a theory of sets. When we use inductive inference to synthesise the operators and rules which make the (enriched) rewriting system terminating and confluent, we are, in

effect, synthesising a theory of 'sets' which is just sufficient to make *dequeue* an eliminator.

The result of applying inductive inference techniques in [JaT 87] is the following rewriting system: a new operator `|*: set set -> set` is introduced and we use the abbreviations `l'` and `l''` for `leaves(dequeue(add(q,d)))` and `d | leaves(q)` resp.

(R1)	`dequeue(eq)`	`->`	`eq`
(R2)	`dequeue(add(eq,d))`	`->`	`eq`
(R3)	`dequeue(add(add(q,d),d))`	`->`	`add(dequeue(add(q,d)),d)`
(R4)	`leaves(add(q,d)`	`->`	`d | leaves (q)`
(R5)	`leaves(eq)`	`->`	`[]`
(R6)	`[] ⊆ S`	`->`	`T`
(R7)	`(d | S) ⊆ []`	`->`	`F`
(Q1)	`d | l'`	`->`	`(d | []) |* l'`
(Q2)	`d | l''`	`->`	`(d | []) |* l''`
(Q3)	`d | (S |* l' )`	`->`	`(d | S) |* l'`
(Q4)	`d | (S |* l'' )`	`->`	`(d | S) |* l''`
(Q5)	`S |* (S |* l')`	`->`	`(S |* S) |* l'`
(Q6)	`S |* (S |* l'')`	`->`	`(S |* S) |* l''`
(S)	`S |* l' ⊆ S |* l''`	`->`	`T`
(T)	`l' ⊆ l''`	`->`	`T`

A specification of 'sets' has been found which ensures that *dequeue* is an eliminator. These 'sets' are more familiarly known as left-associative sequences and *dequeue* is trivially an eliminator because **(T)** is in the equational theory. If we remove **(T)**, then the remaining rewriting system is not terminating and confluent because **(T)** is an instance of **(S)** with `[]` substituted for `S`; i.e. **(T)** is the base case and **(S)** is the inductive case.

Unfortunately, a theory of 'sets' in which **(T)** is not in the equational theory, but in the inductive theory, is our objective. Can we now use the above specification of 'sets' to derive an alternative specification whose equational theory is contained in the usual specification of multisets (i.e. the specification with associative-commutative union) and whose inductive theory contains **(T)**? An examination of the rules **(Q1)**, **(Q2)**, **(Q3)**, **(Q4)**, **(Q5)** and **(Q6)** leads us to conclude that a 'set' constructor which is left-associative is required. The original constructor `_|_` fulfills this requirement and so `_|*_` is not required. The obvious (boolean) relationship between left-associative sequences constructed by `_|_` is the left subsequence relationship. This leads us to conjecture that ⊆ may be specified as the left subsequence operator; this conjecture seems, informally, to be consistent with **(T)**. For example, one can check that

`leaves(dequeue(add(eq,d)))` is a left subsequence of `leaves(add(eq,d))`. This relationship is specified by the following rules:

(R8) `(d | S) ⊆ [] -> F`

(R9) `( d | S) ⊆ ( d1 | S1) -> (d = d1 ) ∧ (S ⊆ S1)`

The equational theory of **(R6)**, **(R8)** and **(R9)** is obviously contained in an associative-commutative theory of multisets of terms of sort *nat*. Moreover, we can easily check that the rewriting systems {**(R1)**,...,**(R6)**}, {**(R1)**,...,**(R6)**,**(R8)**,**(R9)**} and {**(R1)** , ... , **(R6)**,**(R8)**,**(R9)**,**(T)**} are terminating and confluent. **(T)** is not in the equational theory of the second rewriting system, but because the third rewriting system is terminating and confluent, then **(T)** is in the inductive theory of the second system. Thus, in a non-trivial way, we have now shown that *dequeue* is an eliminator.

Of course the theory of left-associative sequences, **(R6)**, **(R8)** and **(R9)**, is not expressive enough to prove the classification of eliminators in all keyless specifications. However, our experience with this example leads us to conjecture that the commutative nature of sets is not necessary for classifying the eliminators of a keyless specification; a suitably expressive specification of finite sequences with associativity as the only permutative property should be sufficient. We note that commutativity may well be required in order to classify the operators in implicitly keyed specifications; for example, commutativity would be required for classifying operators in the specification **Priority_Queue**.

Moreover, when only associativity is required, then we also conjecture that for keyless specifications, inductive inference techniques will always be successful when looking for weakest common generalisations and thus a terminating and confluent rewriting system will be found using these techniques.

2.6.1.2 Theorem Provers

As an alternative to KB completion based rewriting techniques, the classifications may be checked with the aid of a mechanical theorem prover; we have tried out some small examples using the NEVER theorem prover [PaK 87]. This theorem prover uses first-order predicate logic and is based on the ideas contained in the AFFIRM theorem prover [Mus 80b] and the Boyer-Moore theorem prover [BoM 79]. For example, we

were able to prove the above theorem (T) from the usual axioms for **Queue** (interpreted as left-right rules with the rule `front(eq)=error` as NEVER does not have partially ordered sorts), the usual axioms for finite sets with associative-commutative union, the above rules for **leaves**, and some rules for the 'measure' (the analysis technique of [BoM 79] for recursive functions) stating that the ordinal length of `dequeue(t)` is less than the ordinal length of `t`. Unfortunately, this theorem prover is not generally available and so we have been unable to try out more examples.

2.7 Summary

In this chapter we have set up a general framework in which we are able to discuss and analyse the ADT specifications which we aim to implement. We have presented the syntax and semantics of the object ADT specification language, a classification of specifications, and a classification of operators in specifications.

The specification language includes partially ordered sorts (so that error-handling is avoided) and a simple, but adequate hierarchical structuring construct. Specifications are classified according to syntactic criteria as keyless, implicitly keyed, and explicitly keyed. Our principal concern is the implementation of keyless and implicitly keyed specifications wherein operators are classified as generators, selectors, eliminators, rearrangers and other operators. The classification is not syntactic but it depends on the behaviour of the operations in the model of the specification; the classification is compared with two other operator classifications from the literature. We give some examples of how operator classifications are checked manually and we also explore the possibilities of automating the classification using term rewriting systems and semi-automatic theorem provers. Although we are not yet successful in automating the classification, we are hopeful that the outstanding problems will be resolved in the near future.

Chapter Three
Storage Relations

In this chapter we use the partition of the specification to impose more structure on the initial algebra. We define the family of *storage relations* of an ADT. These relations are derived from a specification and describe the order in which the primitive terms "held" in a term of the derived type may be retrieved, or observed. We examine some properties of these relations which are important from an implementation point of view and we use the properties to classify ADTs .

3.1 Relations

Each storage relation describes a relation on primitive sorted terms. A storage relation is not the sub-term relation which describes how terms are constructed; instead, it is a relation which describes the way in which the "data", the primitive sorted terms, are removed and selected from a term of the derived sort, the "data structure". Each storage relation reflects an "observational" view, with respect to the data, of the ADT; namely, a view of how data is stored, manipulated and retrieved in an element of the ADT. We define the storage relations to be the relations which deliver the simplest structure whilst allowing efficient retrieval of the stored data. The definition incorporates several decisions about how the data is retrieved and what is important to reflect in the relation because, in general, there are several ways of retrieving a particular term of the primitive sort from a term of the derived sort. We have only a *static* view of the ADT and cannot predict the frequency or context in which the operations will be applied; therefore we must base our decisions on the operator partition. The motivation for our definitions is as follows.

Given a term t of the derived sort, a particular term d, of the primitive sort, may be retrieved by applying various permutations of rearrangers, eliminators, and a selector. Rearrangers and eliminators may be arbitrarily interleaved, but the application of the selector must of course be last. In general, d may be described (if possible) by a term of the form: $\sigma_s(\sigma_n(\ldots(\sigma_1(t))\ldots))$ where $n \geq 0$, $\forall i: 1 \leq i \leq n: \sigma_i \in (\Sigma_r \cup \Sigma_e)$, and $\sigma_s \in \Sigma_s$. In many specifications, for each term t and d

there will be several possible choices for $\sigma_1,...,\sigma_n$. We will consider, for every term t of the derived sort, how the data it contains may be retrieved (by elimination and selection), and how, after the rearrangement of t, the data is retrieved (by elimination and selection). Namely, we restrict our attention to the following possibilities: $\sigma_1 \in (\Sigma_r \cup \Sigma_e)$ and $\forall i{:}1<i\leq n{:}\ \sigma_i \in \Sigma_e$. We note that for all terms of the primitive sort, and all terms of the derived sort, there may be one, many, or no terms describing the retrieval of the primitive term from the derived term according to the above restrictions.

Each storage relation depends on a term of the derived sort and for a given derived term t, the *storage relation at t* is a relation on the (congruence classes containing the) terms describing the retrieval of the primitive terms from t; i.e. it is a relation on the (congruence classes containing the) primitive terms which are accessible from (the congruence classes containing) t.

In later chapters, we will make implementation decisions according to the properties of the storage relations of an ADT; the relation definitions will ensure that the efficiency of selection, (repeated) elimination and (single application of) rearrangement are taken into account. The efficiency of repeated rearrangement is not ensured; if it is important then new operators for repeated rearrangement should be defined.

We proceed to define the storage relation at t. We assume that all definitions refer to keyless or implicitly keyed specifications and we use examples from Appendix Two to illustrate each definition. In order to make the examples more readable, we use the constants given in the specification of **Nat**; i.e. n abbreviates the n^{th} successor of 0.

3.1.1 The Elimination Relations

We begin by defining $\downarrow t$, a subset of $(T_{\Sigma,E})_\tau$, and then we define a binary relation $\rightarrow_t$ on $\downarrow t$. For a given term t of the derived sort, $\downarrow t$ contains the (congruence classes containing) the rearrangements of t, and all the terms found by repeated application of the eliminators to t and to the rearrangements of t. The definition of $\downarrow t$ reflects the decision to restrict the interleaving of eliminators and rearrangers when describing the retrieval of a primitive sorted term.

Definition: Let (Σ,E) be a specification and let t be a term of the derived sort.

$$\downarrow t =_{def} \{ C \in (T_{\Sigma,E})_\tau \mid \exists n \geq 0: \exists \sigma_1,\ldots,\sigma_n \in \Sigma_\theta: \exists t' \in (T_\Sigma)_\tau:$$
$$((t \equiv_E t') \vee (\exists \sigma_r \in \Sigma_r: t' \equiv_E \sigma_r\, t)) \wedge \sigma_n(\ldots(\sigma_1(t'))\ldots) \in C\} \ ♥$$

We may paraphrase $\downarrow t$ by saying that if t is a term with sort *structure*, then $\downarrow t$ contains the (congruence classes containing the) rearrangements of t and all the valid "sub"-*structures* (with respect to the specification) of t and its rearrangements. Some examples illustrate this definition.

Example 3.1: Consider the **Stack** example and let t be the term

```
push(push(push(create,0),1),2).
```

Then,

$\downarrow t =$ { [push(push(push(create,0),1),2)], [push(push(create,0),1)], [push(create,0)], [create] }.

$\downarrow t$ consists of the classes containing the "sub"-stacks of t. In this example, the "sub"-stacks are also the sub-terms of t.

Example 3.2: Consider the **Queue** example and let t be the term

```
add(add(add(eq,0),1),2).
```

Then,

$\downarrow t =$ { [add(add(add(eq,0),1),2)], [add(add(eq,1),2)], [add(eq,2)], [eq] }.

$\downarrow t$ consists of the classes containing the "sub"-queues of t. In this example the "sub"-queues are not sub-terms of t.

♥ Strictly speaking, only those $\sigma_r(t)$ and $\sigma_n(\ldots(\sigma_1(t'))\ldots)$ which are well-typed are considered. For example, if there is a $s_r \in \Sigma_r$ with arity $\phi \rightarrow \phi$, then $\downarrow t$ contains classes of the form $[\sigma_n(\ldots(\sigma_1(t))\ldots)]$ and $[\sigma_n(\ldots(\sigma_1(t'))\ldots)]$, where $t' \in (T_\Sigma)_\phi$ and $t' \equiv_E \sigma_r(t)$.

Example 3.3: Consider the **Sequence** example and let t be the term

conc(conc(m(0),m(1)),conc(m(2),m(3))).

Then,

$\downarrow t =$ {[conc(conc(m(0),m(1)),conc(m(2),m(3)))],
[conc(conc(m(0),m(1)),m(2))], [conc(m(1),conc(m(2),m(3)))],
[conc(m(2),m(3))], [conc(m(0),m(1))], [conc(m(1),m(2))],
[m(2)], [m(3)], [m(1)], [m(0)], [eseq]}.

$\downarrow t$ consists of the classes containing the "sub"-sequences of t.

Next, a binary relation, $\rightarrow_t$, is defined on the classes in $\downarrow t$. $\rightarrow_t$ is called the *elimination* relation and we paraphrase it as follows: when t has sort *structure*, C and C' are members of $\downarrow t$, and C' is a "sub"-*structure* (with respect to the specification) of C, then $C \rightarrow_t C'$.

Definition: Let (Σ,E) be a specification and let t be a term of the derived sort. The ***elimination relation*** $\rightarrow_t$ is the binary relation on $\downarrow t$ defined by:

$C \rightarrow_t C'$ iff $\exists \sigma \in \Sigma_e: \sigma(C) = C'$ for $C,C' \in \downarrow t$. ♥

As an example, in the **Stack** specification we can show that for

$t =_{def}$ push(push(push(create,0),1),2)

[push(push(create,0),1)] $\rightarrow_t$ [push(create,0)]

because

pop[push(push(create,0),1)] = [push(create,0)],

[push(push(create,0),1)] $\in \downarrow t$, [push(create,0)] $\in \downarrow t$ and *pop* is an eliminator.

As before, we note that $\rightarrow_t$ does not denote the (syntactic) sub-term relation, although it may coincide with it in some specifications such as **Stack**. In the **Queue**

♥ As before, this assumes the convention that $\sigma(C)$ is well-typed.

example, $\rightarrow_t$ looks quite different: for example, we can show that when

$$t =_{def} \text{add}(\text{add}(\text{add}(\text{eq}, 0), 1), 2),$$

then

$$[\text{add}(\text{add}(\text{eq}, 1), 2)] \rightarrow_t [\text{add}(\text{eq}, 2)]$$

because

$$\text{dequeue}[\text{add}(\text{add}(\text{eq}, 1), 2)] = [\text{add}(\text{eq}, 2)],$$

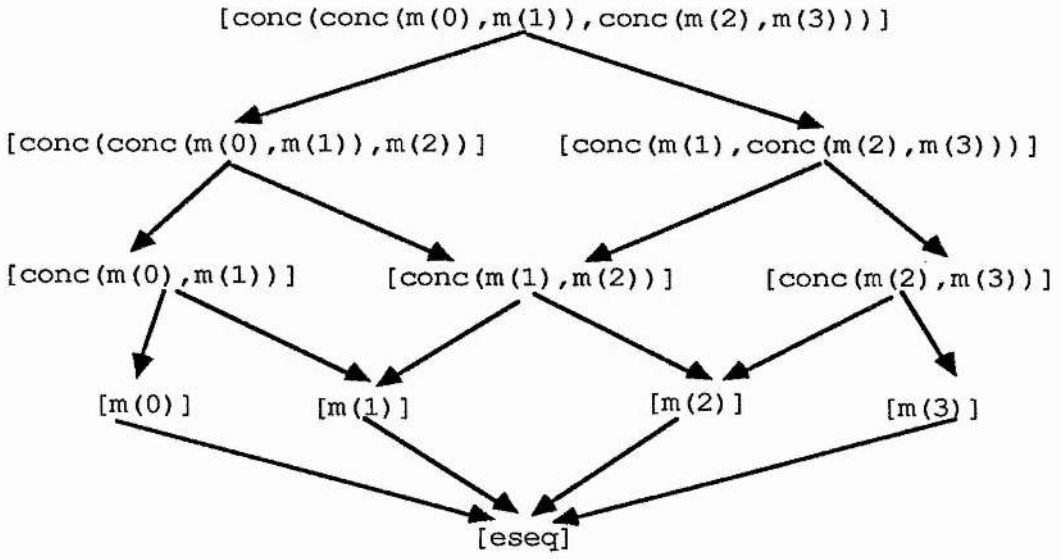
$[\text{add}(\text{add}(\text{eq}, 1), 2)] \in \downarrow t$, $[\text{add}(\text{eq}, 2)] \in \downarrow t$, and *dequeue* is an eliminator.

We note that $\rightarrow_t$ is defined by <u>all</u> the eliminators in a specification and so $(\rightarrow_t)^*$, the reflexive, transitive closure of $\rightarrow_t$, is not, in general, confluent. If there are no rearrangers in the specification then the range of $\rightarrow_t$ is just the set $\{[t'] \mid [t] (\rightarrow_t)^* [t']\}$.

Example 3.4: Consider the **Sequence** example again and let t be the term

$$\text{conc}(\text{conc}(\text{m}(0), \text{m}(1)), \text{conc}(\text{m}(2), \text{m}(3))).$$

The elimination relation, represented as a directed graph, is the following:



3.1.2 The Storage Relations

The selectors and the elimination relation are used to define a family of relations on elements of the primitive sort. The family is indexed by the derived sort terms. Each

term t of the derived sort determines a set of primitive-sorted classes $\Downarrow t$, called the *contents set* of t, and a binary relation $\Rightarrow_t$ on the contents set, called the *storage relation* at t. The storage relation is induced, in a natural way, from the elimination relation by the selectors. We use the notation $f^{->}(S)$ for the image of the set S under the operation f. If R is a binary relation on S, then we also use the notation $f^{->}(R)$ for the relation P defined by

$$xPy \text{ iff } (\exists s,s' \in S)\ ((sRs') \wedge (f(s) = x) \wedge (f(s') = y)).$$

Definition: Let (Σ,E) be a specification and let t be a term of the derived sort. The ***contents set*** of t, $\Downarrow t$, is the set of primitive sorted classes defined as follows:

$$\Downarrow t =_{def} \bigcup \{ \sigma^{->}(\downarrow t)_\alpha \mid (\sigma \in \Sigma_s) \wedge ((\alpha->\delta) \in T_\sigma) \}$$

Definition: Let (Σ,E) be a specification and let t be a term of the derived sort. The ***storage relation*** at t, $\Rightarrow_t$, is the binary relation on the contents set of t defined by

$$\Rightarrow_t =_{def} \bigcup \{ \sigma^{->}(\rightarrow_t)_\alpha \mid (\sigma \in \Sigma_s) \wedge ((\alpha->\delta) \in T_\sigma) \}$$

The interpretation of $\Rightarrow_t$, like that of $\rightarrow_t$, depends on the specification. For example, in the **Stack** specification, $\Rightarrow_t$ denotes "after"; $x \Rightarrow_t y$ means that x was put on the stack t after y and is therefore more accessible. On the other hand, in the **Queue** specification, $\Rightarrow_t$ denotes the converse, i.e. "before"; $x \Rightarrow_t y$ means that x was put on the queue t before y and is therefore more accessible.

We illustrate these definitions with some examples. In the examples, we use only ground generator terms which contain distinct sub-terms of the primitive sort; we will discuss this further in §3.2.

Example 3.5: Consider the **Stack** example and let t be the term

push (push (push (push (create, 0) , 1) , 2) , 3).

The contents set $\Downarrow t$ and the storage relation $\Rightarrow_t$ at t are as follows:

$$\Downarrow t = \{ [3], [2], [1], [0] \}$$

$$[3] \Rightarrow_t [2]$$

$$[2] \Rightarrow_t [1]$$

$$[1] \Rightarrow_t [0]$$

which we represent as a directed graph as follows:

Example 3.6: Consider the **Queue** example and let t be the term

add (add (add (add (eq, 0) , 1) , 2) , 3).

The contents set $\Downarrow t$ and the storage relation $\Rightarrow_t$ at t are as follows:

$$\Downarrow t = \{ [3], [2], [1], [0] \}$$

$$[2] \Rightarrow_t [3]$$

$$[1] \Rightarrow_t [2]$$

$$[0] \Rightarrow_t [1]$$

represented graphically as:

Example 3.7: Consider the **Sequence** example and let t be the term

conc(conc(m(0),m(1)),conc(m(2),m(3))).

The contents set $\Downarrow t$ and the storage relation $\Rightarrow_t$ at t are as follows:

$$\Downarrow t = \{ [3], [2], [1], [0] \}$$

$$[0] \Rightarrow_t [0] \qquad [2] \Rightarrow_t [2]$$

$$[1] \Rightarrow_t [1] \qquad [3] \Rightarrow_t [3]$$

$$[0] \Rightarrow_t [1] \qquad [1] \Rightarrow_t [2]$$

$$[2] \Rightarrow_t [3] \qquad [3] \Rightarrow_t [2]$$

$$[2] \Rightarrow_t [1] \qquad [1] \Rightarrow_t [0]$$

represented graphically as:

The contents sets and storage relations can reflect (seemingly) minor changes and additions to familiar specifications.

Example 3.8: Consider the **List** example and let t be the term

3:(2:(1:(0:nil))).

The contents set $\Downarrow t$ and the storage relation $\Rightarrow_t$ at t are as follows:

$$\Downarrow t = \{ [3], [2], [1], [0] \}$$

$$[3] \Rightarrow_t [2]$$

$$[2] \Rightarrow_t [1]$$

$$[1] \Rightarrow_t [0]$$

represented graphically as:

Example 3.9: Consider the **`Circular_List_right`** example and let t be the term

```
3:(2:(1:(0:nil))).
```

The contents set $\Downarrow t$ and the storage relation $\Rightarrow_t$ at t are as follows:

$$\Downarrow t = \{ [3], [2], [1], [0] \}$$

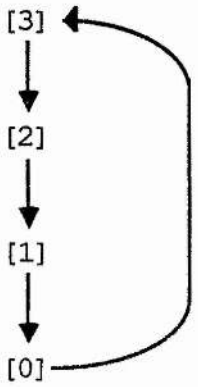
$$[3] \Rightarrow_t [2]$$

$$[2] \Rightarrow_t [1]$$

$$[1] \Rightarrow_t [0]$$

$$[0] \Rightarrow_t [3]$$

represented graphically as:



Example 3.10: Consider the **`Reversible_List_1`** example (where a hidden append operator is used to define the reverse operator) and let t be the term

```
3:(2:(1:(0:nil))).
```

The contents set $\Downarrow t$ and the storage relation $\Rightarrow_t$ at t are as follows:

$$\Downarrow t = \{ [3], [2], [1], [0] \}$$

$[0] \Rightarrow_t [1]$ $\quad$ $[1] \Rightarrow_t [2]$

$[2] \Rightarrow_t [3]$ $\quad$ $[3] \Rightarrow_t [2]$

$[2] \Rightarrow_t [1]$ $\quad$ $[1] \Rightarrow_t [0]$

represented graphically as:

We have the same results when considering a different specification of reversible lists.

Example 3.11: Consider the **`Reversible_List_2`** example (where a hidden "head-collecting" operator is used to define the reverse operator) and let t be the term

`3:(2:(1:(0:nil))).`

The contents set $\Downarrow t$ and the storage relation $\Rightarrow_t$ at t are as follows:

$\Downarrow t = \{ [3], [2], [1], [0] \}$

$[0] \Rightarrow_t [1]$ $\quad$ $[1] \Rightarrow_t [2]$

$[2] \Rightarrow_t [3]$ $\quad$ $[3] \Rightarrow_t [2]$

$[2] \Rightarrow_t [1]$ $\quad$ $[1] \Rightarrow_t [0]$

represented graphically as:

Of course modifying a specification need not change the contents sets and storage relations.

Example 3.12: Consider the **Binary_Tree** and **SBinary_Tree** examples. Let t be the term

$$\text{comb}(\text{comb}(\text{n}(0),\text{n}(1)),\text{n}(2)).$$

The addition of the rearranger *switch* does not alter the storage relation; in both examples we have the following contents set and storage relation at t:

$$\Downarrow t = \{[2], [1], [0], [0+1], [(0+1)+2]\}$$

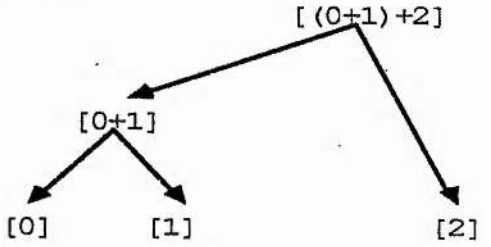
$$[0+1] \Rightarrow_t [0]$$

$$[0+1] \Rightarrow_t [1]$$

$$[(0+1)+2] \Rightarrow_t [0+1]$$

$$[(0+1)+2] \Rightarrow_t [2]$$

represented graphically as:



We should note that the contents set does not always return the same results as the **leaves** function. Consider the **Stack2** specification with t as the term given in the **Stack** example 3.5, namely push(push(push(push(create,0),1),2),3). In the **Stack2** specification the contents set $\Downarrow t$ is a subset of the contents set given in 3.5 since $\Downarrow t=\{[3],[1]\}$; also, the storage relation is weaker than the storage relation given in 3.5 since $x \Rightarrow_t y$ is only true for x= [3], y= [1].

3.1.3 Finite Storage Relations

It is useful to know when the contents sets, and hence the storage relations, of an ADT are finite.

Lemma: Let (Σ,E) be a keyless/implicitly keyed specification. If E does not contain non-permutative equations between generator terms, then for all terms t of the derived sort, the contents set of t, $\Downarrow t$, is finite.

proof: If there are no non-permutative equivalences between generator terms, then **leaves** is well-defined on $T_{\Sigma,E}$. When **leaves** is well-defined, then by the definition of the partitioned signature, every eliminator operator $e:\alpha \rightarrow \tau$, where $\alpha \in \{\phi,\tau\}$, has the following property:

$$(\forall C \in (T_{\Sigma,E})_\alpha) \quad (\textbf{leaves}(e(C)) = \textbf{leaves}(C) \vee \textbf{leaves}(e(C)) \subset \textbf{leaves}(C))).$$

Using this property, we can show that for all α-sorted terms t, the set

$$t' =_{def} \{[e^n(t)]] \mid n \in N\}$$

is finite, and therefore the set

$$t'' =_{def} \{[E(t)]] \mid E \in (\Sigma_e)^*\} \quad \blacktriangledown$$

is finite.

Every signature is finite, and so when (Σ,E) has n eliminators, $e_1,...,e_n$, m rearrangers, $r_1,...,r_m$, and p selectors, $s_1,...,s_p$, then

$$\downarrow t = t'' \cup (r_1(t))'' \cup ... \cup (r_m(t))''. \quad \blacktriangledown$$

$\downarrow t$ is clearly finite, and therefore the contents of t,

$$\Downarrow t = s_1^{->}(\downarrow t) \cup ... \cup s_p^{->}(\downarrow t) \quad \blacktriangledown$$

is also finite.

▼As before, we assume that all terms are well-typed.

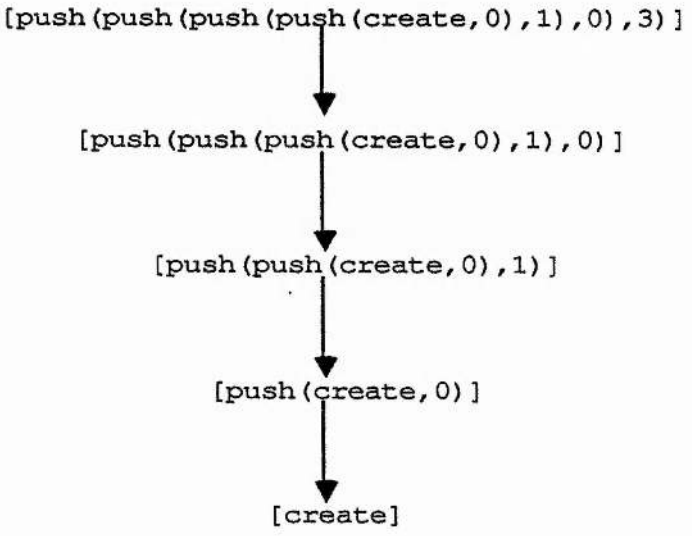
3.2 Classifying ADTs by Storage Relations

An informal examination of the example storage relations in the previous section gives us a good idea of the behaviour of the ADTs concerned; it is easy to extrapolate from that behaviour and suggest ways in which the ADTs might be implemented. In this section we formalise that behaviour by considering the properties which hold for the class of storage relations determined by some of the (derived sort) terms of an ADT.

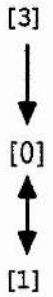
We do not consider the storage relations determined by *all* of the (derived sort) terms of an ADT for the simple reason that terms of the primitive sort are being regarded as place holders in terms of the derived type. Namely, we are not concerned with the *values* of the terms of the primitive sort as such, but with their *positions* in the ground generator terms of the derived sort. In the preceding section we have used examples in which the primitive sorted sub-terms occurring in the derived sort ground generator terms are incomparable. Consider the following example from the **Stack** specification:

$$\text{let } t =_{def} \texttt{push(push(push(push(create,0),1),0),3)}.$$

The elimination relation $\rightarrow_t$ on $\downarrow t$ is antisymmetric:



But the storage relation at t, $\Rightarrow_t$ is not antisymmetric:



Of course the elimination relation and storage relation do not, in general, have the same structure, but in this case the congruence between the innermost primitive term in t, 0, and the second outermost primitive term in t, 0, changes the properties that the storage relation would otherwise have. In this case, the property of antisymmetry no longer holds as it did in Example 3.5.

We need, at least, to restrict our attention to those classes which contain ground generator terms which themselves contain discrete (i.e. incomparable) primitive sorted sub-terms. We shall refer to these classes as the *restricted* classes and to the relations on those classes as the *restricted* relations.

There is some difficulty handling restrictions of this kind elegantly within the algebraic framework because terms are freely constructed. Perhaps the best solution would be to consider a *trace algebra*, where the traces are just the descriptions of the terms which describe the retrieval of the primitive sorted terms from the derived sort terms according to the definitions given in §3.1. Our problems are caused by the fact that in the standard model we cannot "remember" how we retrieved a data item; i.e. we can have the situation that for some, but not all terms t, s(e(e(t))) is identified with s(e(t)), where s is a selector and e is an eliminator. For the purposes of analysing an ADT by storage type, we should only like s(e(e(t))) to be identitifed with s(e(t)) when the identification holds for all ground generator instantiations of t; i.e. when s(e(e(t)))=s(e(t)) is in the inductive theory. We do not pursue the "trace" approach here; instead, we simply do not consider ground generator terms with repeated occurrences of a primitive sorted subterm and thus we avoid the situation described above.

We note that in those specifications where selectors, when applied to ground generator terms, do not return sub-terms of the argument, further restrictions on the structure of the primitive specification may be desirable. We shall discuss this further in chapter 6.

3.2.1 Restricting Classes and Storage Relations

We introduce the "*" notation to refer to the restricted congruence classes and storage relations of a specification.

First we define $T^*_{\Sigma,E}$, the classes of $T_{\Sigma,E}$ which contain ground generator terms with at most one occurrence of each primitive sorted sub-term.

Definition: Let (Σ,E) be a specification.

$$T^*_{\Sigma,E} =_{def} \{C \in T_{\Sigma,E} \mid \exists t \in C: t \in T_{\Sigma g} \wedge t \text{ contains at most one occurrence of each } \delta\text{-sorted sub-term}\}$$

Some examples will illustrate this definition.

In the **Queue** specification, $T^*_{\Sigma,E}$ contains classes such as [add(eq,0)] and [dequeue(add(add(eq,0),0))], but it does not contain the class [add(add(eq,0),0)]. Some operations (in the *generator* and *others* partition) may become partial on the elements of $T^*_{\Sigma,E}$. For example, in the **Sequence** specification, $T^*_{\Sigma,E}$ contains the classes [conc(m(0),m(1))] and [conc(m(1),m(0))], but it does not contain the class [conc(conc(m(0),m(1)),conc(m(1),m(0)))]; i.e. the function defined by *conc* is partially defined on $T^*_{\Sigma,E}$.

We do not insist that *all* ground generator terms in the classes contain only one occurrence of each primitive term. This would be too restrictive and could exclude all the classes from some specifications with non-permutative equations between generator terms. It is not easy to imagine many useful specifications containing non-permutative equations between generators, but here is an example of one such specification: the specification of a "Von Neumann" variable.

```
spec      VNvariable
basedon   Nat
sorts     sto, fsto
subsorts  fsto ≤ sto
gen       e        : sto
          update   : sto nat -> fsto
ops       lookup   : fsto -> nat
eqns
∀s:fsto,i:nat.      lookup(update(s,i)) = i                R1
∀s,s':sto,i:nat.    update(s,i) = update(s',i)             E1
end
```

In this example, $T^*_{\Sigma,E} = T_{\Sigma,E}$. However, $T^*_{\Sigma,E}$ would be empty if we insisted that *every* ground generator term contained discrete primitive sorted sub-terms, because

$$\text{update}(e,i) \equiv_E \text{update}(\text{update}(e,i),i) \quad \text{for } i \in \text{Nat}.$$

Second, we define the (restricted) elimination relation $\rightarrow_t^*$ and the (restricted) storage relation $\Rightarrow_t^*$ on $(T^*_{\Sigma,E})_\tau$ and $(T^*_{\Sigma,E})_\delta$ resp. These relations differ from the definitions given in §3.1.1 and §3.1.2 only in their domains; the notation should not be confused with the reflexive, transitive closure which is denoted by $(\rightarrow_t)^*$ and $(\Rightarrow_t)^*$ resp.

Definition: Let (Σ,E) be a specification and let t be a term of the derived sort.

$$\downarrow t^* =_{def} \{ C \in (T^*_{\Sigma,E})_\tau \mid \exists n \geq 0: \exists \sigma_1,\ldots,\sigma_n :\in \Sigma_e \; \exists t' \in (T_\Sigma)_\tau: ((t \equiv_E t') \vee (\exists \sigma_r \in \Sigma_r: t' \equiv_E t)) \wedge \sigma_n(\ldots(\sigma_1(t'))\ldots) \in C \}$$ ♥

$$C \rightarrow_t^* C' =_{def} \exists \sigma \in \Sigma_e: \sigma(C) = C' \quad \text{for } C,C' \in \downarrow t.$$ ♥

$$\Downarrow t^* =_{def} \cup \{ \sigma^{->}(\downarrow t^*)_\alpha \mid (\sigma \in \Sigma_s) \wedge ((\alpha->\delta) \in T_\sigma) \}$$

$$\Rightarrow_t^* =_{def} \cup \{ \sigma^{->}(\rightarrow_t^*)_\alpha \mid (\sigma \in \Sigma_s) \wedge ((\alpha->\delta) \in T_\sigma)$$

♥ As before, we assume that all terms are well-typed.

3.2.2 Properties of Relations

In the following chapters we will make implementation decisions based on properties of the structures defined by the restricted storage relations. In this section, we define some of those properties. Of course, the implementer must use his or her imagination to decide which properties might be useful for implementation purposes and our list is not exhaustive.

Various conditions may be imposed on a relation on a set and its elements; the following conditions from [End 77] and [Hal 67] are standard: ***reflexive, transitive, symmetric, antisymmetric, comparable, least, minimal, maximal*** and ***greatest***. We define some further conditions. In the following, we use the notation $\underline{y}(n)$ to denote $\langle y_1,...,y_n \rangle$ such that $(\forall i,j:1 \leq i,j \leq n)\ (y_i = y_j) \Rightarrow (i = j)$.

Definition:

1. When R is a relation, ***(R)**** is the reflexive, transitive closure of R.

2. A relation R on set S is ***down-directed*** iff every pair of elements in S has an upper bound in (R)*: i.e.

$$(\forall x,y \in S)\ (\exists w \in S)\ (w(R)^*x \wedge w(R)^*y).$$

3. A relation on set S is ***upwards-directed*** iff every pair of elements in S has a lower bound in (R)*; i.e. $(\forall x,y \in S)\ (\exists w \in S)\ (x(R)^*w \wedge y(R)^*w)$.

4. A relation R on set S is ***n-regular*** iff when R is considered as directed graph, every node has outdegree 0 or outdegree n; i.e.

$$(\forall x \in S)$$
$$(\sim(\exists z \in S: xRz) \vee$$
$$((\exists \underline{y}(n):\ xRy_1 \wedge ... \wedge xRy_n) \wedge \sim(\exists z \in S \setminus \{y_1,...,y_n\}:\ xRz)).$$

5. A relation R on set S is ***(n:m)-regular*** iff when R is considered as directed graph, every node has outdegree 0 or outdegree p where $n \leq p \leq m$; i.e.

$(\forall x \in S)$

$(\sim(\exists z \in S : xRz) \vee$

$((\exists p : n \leq p \leq m)$

$(\exists \underline{y}(p): xRy_1 \wedge \ldots \wedge xRy_p) \wedge \sim(\exists z \in S \backslash \{y_1,\ldots,y_p\}: xRz)))$.

6. A relation R on set S is ***singly-linked linear*** iff R is 1-regular, $(R)^*$ is antisymmetric, all pairs in $(R)^*$ are comparable, and when S is not empty, minimal and maximal elements exist.

7 A relation R on set S is ***doubly-linked*** iff R is (1:2)-regular and symmetric, and all pairs in S are comparable by $(R)^*$.

8. A relation R on set S is ***doubly-linked linear*** iff R is doubly-linked and when R is considered as a directed graph and $|S|>2$, there are exactly two nodes with outdegree 1.

9. A relation R on S is ***singly-linked circular*** iff R is 1-regular and 1-regular and antisymmetric, all pairs in S are comparable by $(R)^*$, and $(R)^*$ is symmetric.

10. A relation R on S is ***doubly-linked circular*** iff R is doubly-linked and when $|S|>2$, R is 2-regular.

11. A relation R on S is ***singly-linked down-directed*** iff R is down-directed and $(R)^*$ is antisymmetric.

12. A relation R on S is ***unstructured*** iff R is empty.

13. A relation R on S is ***multi-access*** iff R is transitive.

14. A relation R on S is ***random-access*** iff R is multi-access, all pairs in S are comparable, and R is symmetric.

We use the properties of the (restricted) storage relations of an ADT to define a classification of ADTs:

Definition: Let (Σ,E) be a specification. When for every term t of the derived sort, the relation $\Rightarrow_t^*$ has the property *P* (of being singly-linked linear etc.) we say that (Σ,E) has ***storage type P***.

3.3 Some Example ADT Classifications

In this section we give a classification of some example ADTs. There may be several ways to classify each ADT and in the following chapters we discuss the significance of particular classifications. The proofs of the classifications are sketched informally and statements marked by ♦ will proven in Appendix Three. In the examples, the relations $\rightarrow_t^*$ and $\Rightarrow_t^*$ are represented by sets of ordered pairs; we use standard set notation with "\" for set difference.

Example 3.13: The **Stack** specification has singly-linked linear storage type.

Informal Proof:

Let $n \in N$ and let $\mathtt{push}^{n+1}(x_1,\ldots,x_{n+1})$ abbreviate

$$\mathtt{push}(\mathtt{push}(\ldots\mathtt{push}(\mathtt{create},x_1),\ldots,x_n),x_{n+1}).$$

Every (restricted) term of sort stack has normal form create or $\mathtt{push}^{n+1}(x_1,\ldots,x_{n+1})$, where $x_1,\ldots,x_{n+1} \in \mathtt{nat}$, and

$$(\forall i,j{:}1\leq i,j\leq n+1)\ \ x_i \equiv_E x_j \Rightarrow i = j.$$

Any condition holds for the empty relation $\Rightarrow_{\mathtt{create}}^*$ and so we need only consider terms of sort nes. We classify the storage type by induction.

lemma: $\Rightarrow_t^*$, where $t =_{def} \mathtt{push}^{n+1}(x_1,\ldots,x_{n+1})$, is singly-linked linear.

proof:

Base case: (n=0)

$t = \mathtt{push}(\mathtt{create},x_1)$.

$\downarrow t^* = \{[\mathtt{push}(\mathtt{create}, x_1)], [\mathtt{create}]\}$

$\rightarrow_t^* = \{<[\mathtt{push}(\mathtt{create}, x_1)], [\mathtt{create}]>\}$

$\Downarrow t^* = \{[x_1]\}$

$\Rightarrow_t^* = \{\}$

$\Rightarrow_t^*$ is trivially singly-linked linear.

Induction step: Assume $n \in N$, and $\Rightarrow_t^*$, where $t =_{def} \mathtt{push}^{n+1}(x_1, \ldots, x_{n+1})$, is singly-linked linear.

Consider $s =_{def} \mathtt{push}(t, y)$. By definition and induction,

$\downarrow s^* = \downarrow t^* \cup \{[\mathtt{push}(t, y)]\}$ ♥

$\rightarrow_s^* = \rightarrow_t^* \cup \{<[\mathtt{push}(t, y)], [t]>\}$ ♥

$\Downarrow s^* = \Downarrow t^* \cup \{[y]\}$ ♥

$\Rightarrow_s^* = \Rightarrow_t^* \cup \{<[y, x_{n+1}]>\}$ ♥

When $\Rightarrow_t^*$ is 1-regular, $(\Rightarrow_t^*)^*$ is antisymmetric and all pairs in $(\Rightarrow_t^*)^*$ are comparable, then $\Rightarrow_t^* \cup \{<[y, x_{n+1}]>\}$ also fulfills these conditions. The minimal element is $[x_1]$ and the maximal element is $[y]$.

Conclusion: **Stack** has singly-linked linear storage type. □

Example 3.14: The **Queue** specification has singly-linked linear storage type.

Informal Proof:

Let $n \in N$ and let $\mathtt{add}^{n+1}(x_1, \ldots, x_{n+1})$ abbreviate

$$\mathtt{add}(\mathtt{add}(\ldots\mathtt{add}(eq, x_1), \ldots, x_n), x_{n+1}).$$

Every (restricted) term of sort *queue* has normal form $\mathtt{eq}$ or $\mathtt{add}^{n+1}(x_1, \ldots, x_{n+1})$, where $x_1, \ldots, x_n \in \mathtt{nat}$, and $(\forall i, j : 1 \leq i, j \leq n+1)\ x_i =_E x_j \Rightarrow i = j$. Any condition holds for the empty relation $\Rightarrow_{eq}^*$ and so we need only consider terms of sort *neq*. We classify the storage type by induction.

lemma: $\Rightarrow_t^*$, where $t =_{def} add^{n+1}(x_1, \ldots, x_{n+1})$, is singly-linked linear.

proof:

Base case: (n=0)

$t = add(add, x_1)$

$\downarrow t^* = \{[add(eq, x_1)], [eq]\}$

$\rightarrow_t^* = \{<[add(eq, x_1), eq]>\}$

$\Downarrow t^* = \{x_1\}$

$\Rightarrow_t^* = \{\}$

$\Rightarrow_t^*$ is trivially singly-linked linear.

Induction step: Assume $n \in N$ and $\Rightarrow_t^*$, where $t =_{def} add^{n+1}(x_1, \ldots, x_{n+1})$, is singly-linked linear.

Consider $s =_{def} add(t, y)$. By definition and induction,

$\downarrow s^* =$

$\{[add^{n+2}(x_1, \ldots, x_{n+1}, y)], [add^{n+1}(x_2, \ldots, x_{n+1}, y)], \ldots, [add^1(y)], [eq]\}$ ♥

$\rightarrow_s^* = \{<[add^{n+2}(x_1, \ldots, x_{n+1}, y)], [add^{n+1}(x_2, \ldots, x_{n+1}, y)]>,$

$<[add^{n+1}(x_2, \ldots, x_{n+1}, y)], [add^n(x_3, \ldots, x_{n+1}, y)]>$

$\ldots,$

$<[add^1(y)], [eq]>\}$ ♥

$\Downarrow s^* = \Downarrow t^* \cup \{[y]\}$ ♥

$\Rightarrow_s^* = \Rightarrow_t^* \cup \{<[x_{n+1}], [y]>\}$ ♥

When $\Rightarrow_t^*$ is 1-regular, $(\Rightarrow_t^*)^*$ is antisymmetric and all pairs in $(\Rightarrow_t^*)^*$ are comparable, then $\Rightarrow_t^* \cup \{<[x_{n+1}], [y]>\}$ also fulfills these conditions. The minimal element is $[y]$ and the maximal element is $[x_1]$.

Conclusion: **Queue** has singly-linked linear storage type. □

Example 3.15: The **Circular_List_right** specification has singly-linked circular storage type.

Informal Proof:

Let $n \in N$ and let $(x_{n+1}:\ldots:x_1)$ abbreviate $x_{n+1}:(x_n:(\ldots(x_1:nil)\ldots))$. Every (restricted) term of sort has normal form nil or $x_{n+1}:(x_n:(\ldots(x_1:nil)\ldots))$, where $x_1,\ldots,x_n \in nat$, and $(\forall i,j:1\leq i,j\leq n+1)\ x_i =_E x_j \Rightarrow i=j$. Any condition holds for the empty relation $\Rightarrow_{nil}^*$ and so we need only consider terms of sort nel. This example is more complicated than the previous ones and so before classifying the storage type, we give some lemmas.

lemma 1: $shift(x_{n+1}:\ldots:x_1) \equiv x_n:\ldots:x_1:x_{n+1}$ ♥

lemma 2: $\Downarrow(x_{n+1}:\ldots:x_1)^* = \{[x_{n+1}],[x_n],\ldots,[x_1]\}$ ♥

lemma 3: $\Rightarrow(x_{n+1}:\ldots:x_1)^* =$

$$\{<[x_{n+1}],[x_n]>,<[x_n],[x_{n-1}]>,\ldots,<[x_2],[x_1]>\} \cup \{[<[x_1],[x_{n+1}]>]\}$$ ♥

Now we classify the storage type by induction.

lemma: $\Rightarrow_t^*$, where $t =_{def} (x_{n+1}:\ldots:x_1)$, is singly-linked circular.

proof:

Base case: (n=0)

$t = x_1$

$\Downarrow t^* = \{[x_1]\}$

$\Rightarrow_t^* = \{\}$

$\Rightarrow_t^*$ is trivially singly-linked circular.

Induction step: Assume $n \in N$ and $\Rightarrow_t^*$, where $t =_{def} (x_{n+1}:\ldots:x_1)$, is singly-linked circular.

Consider $s =_{def} y:t$. By lemmas 1-3 we can show that

$\Downarrow s^* = \Downarrow t^* \cup \{[y]\}$ ♥

$\Rightarrow_s{}^* = (\Rightarrow_t{}^* \cup \{<[y_n],[x_{n+1}]> <[x_1],[y]>\}) \setminus \{<[x_1],[x_{n+1}]>\}$ ♥

When $\Rightarrow_t{}^*$ is 1-regular and antisymmetric, and $(\Rightarrow_t{}^*)^*$ is symmetric and all pairs in $(\Rightarrow_t{}^*)^*$ are comparable, then $(\Rightarrow_t{}^* \cup \{<[y_n],[x_{n+1}]> <[x_1],[y]>) \setminus \{<[x_1],[x_{n+1}]>\}$ also fulfills these conditions.

Conclusion: **`Circular_List_right`** has singly-linked circular storage type. □

Example 3.16: The **`Circular_List_left`** specification has singly-linked circular storage type.

Informal Proof:

Let $n \in N$ and let $(x_{n+1}:\ldots:x_1)$ abbreviate $x_{n+1}:(x_n:(\ldots(x_1:nil)\ldots))$. Every (restricted) term of sort has normal form `nil` or $x_{n+1}:(x_n:(\ldots(x_1:nil)\ldots))$, where $x_1,\ldots,x_n \in nat$, and $(\forall i,j:1\leq i,j\leq n+1)\ x_i =_E x_j \Rightarrow i=j$. Any condition holds for the empty relation $\Rightarrow_{nil}{}^*$ and so we need only consider terms of sort *nel*. Before classifying the storage type we give some lemmas.

lemma 1: $last(x_{n+1}:\ldots:x_1) \equiv x_1$ ♥

lemma 2: $rem(x_{n+1}:\ldots:x_1) \equiv x_{n+1}:\ldots:x_2$ ♥

lemma 3: $shift(x_{n+1}:\ldots:x_1) \equiv x_1:x_{n+1}\ldots:x_2$ ♥

lemma 4: $\Downarrow (x_{n+1}:\ldots:x_1)^* = \{[x_{n+1}],[x_n],\ldots,[x_1]\}$ ♥

lemma 5: $\Rightarrow (x_{n+1}:\ldots:x_1)^* =$

$\{<[x_{n+1}],[x_n]>,<[x_n],[x_{n-1}]>,\ldots,<[x_2],[x_1]>\} \cup \{[<[x_1],[x_{n+1}]>\}$ ♥

Now we classify the storage type by induction.

lemma: $\Rightarrow_t^*$, where $t =_{def} (x_{n+1}:\ldots:x_1)$, is singly-linked circular.

proof:

Base case: (n=0)

$t = x_1$

$\Downarrow t^* = \{[x_1]\}$

$\Rightarrow_t^* = \{\}$

$\Rightarrow_t^*$ is trivially singly-linked circular.

Induction step: Assume $n \in N$ and $\Rightarrow_t^*$, where $t =_{def} (x_{n+1}:\ldots:x_1)$, is singly-linked circular.

Consider $s =_{def} y:t$. By lemmas 1-5 we can show that

$\Downarrow s^* = \Downarrow t^* \cup \{[y]\}$ ♥

$\Rightarrow_s^* = (\Rightarrow_t^* \cup \{<[y_n], [x_{n+1}]> <[x_1], [y]>\}) \setminus \{<[x_1], [x_{n+1}]>\}$ ♥

When $\Rightarrow_t^*$ is 1-regular and antisymmetric, and $(\Rightarrow_t^*)^*$ is symmetric and all pairs in $(\Rightarrow_t^*)^*$ are comparable, then $(\Rightarrow_t^* \cup \{<[y_n], [x_{n+1}]> <[x_1], [y]>\}) \setminus \{<[x_1], [x_{n+1}]>\}$ also fulfills these conditions.

Conclusion: **Circular_List_left** has singly-linked circular storage type. □

Example 3.17: The **Reversible_List_1** specification has doubly-linked linear storage type.

Informal Proof:

Let $n \in N$ and let $(x_{n+1}:\ldots:x_1)$ abbreviate $x_{n+1}:(x_n:(\ldots(x_1:nil)\ldots))$. Every (restricted) term of sort has normal form nil or $x_{n+1}:(x_n:(\ldots(x_1:nil)\ldots))$, where $x_1, \ldots, x_n \in nat$, and $(\forall i,j:1\leq i,j\leq n+1)\ x_i \equiv_E x_j \Rightarrow i = j$. Any condition holds for the empty relation $\Rightarrow_{nil}^*$ and so we need only consider terms of sort *nel*. Before classifying the storage type we give some lemmas.

lemma 1: $(\forall y{:}nat)\ app(y:nil, x_{n+1}:\ldots:x_1) \equiv x_{n+1}:\ldots:x_1:y$ ♥

lemma 2: $rev(x_{n+1}:\ldots:x_1) \equiv x_1:\ldots:x_{n+1}$ ♥

lemma 3: $\Downarrow(x_{n+1}:\ldots:x_1)^* = \{[x_{n+1}],[x_n],\ldots,[x_1]\}$ ♥

lemma 4: $\Rightarrow(x_{n+1}:\ldots:x_1)^* = \{<[x_i],[x_{i-1}]>,<[x_j],[x_{j+1}]> \mid 2\leq i\leq n+1, 1\leq j\leq n\}$ ♥

Now we classify the storage type by induction.

lemma: $\Rightarrow_t^*$, where $t =_{def} (x_{n+1}:\ldots:x_1)$, is doubly-linked linear.

proof:

Base case: (n=0)

$t = x_1$

$\Downarrow t^* = \{[x_1]\}$

$\Rightarrow_t^* = \{\}$.

$\Rightarrow_t^*$ is trivially doubly-linked linear.

Induction step: Assume $n \in N$ and $\Rightarrow_t^*$, where $t =_{def} (x_{n+1}:\ldots:x_1)$, is doubly-linked linear.

Consider $s =_{def} y:t$. By lemmas 1-4 we can show that

$\Downarrow s^* = \Downarrow t^* \cup \{[y]\}$ ♥

$\Rightarrow_s^* = \Rightarrow_t^* \cup \{<[y],[x_{n+1}]>, <[x_{n+1}],[y]>\}$ ♥

When $\Rightarrow_t^*$ is (1:2)-regular and symmetric, all pairs in $(\Rightarrow_t^*)^*$ are comparable, and there are only two nodes with outdegree 1, then $\Rightarrow_t^* \cup \{<[y],[x_{n+1}]>, <[x_{n+1}],[y]>\}$ also fulfills these conditions. $[x_1]$ and $[y]$ are the (labels of the) nodes in $\Rightarrow_s^*$ with outdegree 1.

Conclusion: **Reversible_List_1** has doubly-linked linear storage type. □

Example 3.18: The **Sequence** specification has doubly-linked linear storage type.

This specification is difficult to work with because of the associativity of the generator *conc* (as specified by equation R13). We note that the rewrite rules {R1,...,R12} form a confluent, terminating rewrite system. Although R13 is not in the inductive theory of {R1,...,R12}, (we can check this by finding ground terms which are counter examples) there are some models in ***Loose*(Sequence)** in which it is valid. When we add R13 to the equational theory, then the set of rewrite rules {R1,...,R13} is not terminating and confluent because although R13 can be oriented from left to right using the Knuth-Bendix ordering, the completion procedure produces an infinite set of rewrite rules. This infinite set is a semi-decision procedure and using inductionless induction we can show, for example, that the following equations are consistent with {R1,...,R12}:

```
∀t:nes,s,s':seq.
lrem(conc(conc(t,s),s'))    = lrem(conc(t,conc(s,s')))
left(conc(conc(t,s),s'))    = left(conc(t,conc(s,s')))
rrem(conc(conc(s,s'),t))    = rrem(conc(s,conc(s',t)))
right(conc(conc(s,s'),t))   = right(conc(s,conc(s',t)))
```

We must be be able to assume the validity of R13 because the associativity of *conc* is essential to the proof of the classification; without it we would have to quantify over all the seq-sorted terms.

Informal Proof:

Let $n \in N$ and let $sq(x_1:\ldots:x_{n+1})$ abbreviate

$$conc(m(x_1),conc(m(x_2),\ldots,conc(m(x_n),m(x_{n+1})\ldots)).$$

Every (restricted) equivalence class contains a term of the form eseq or $sq(x_1:\ldots:x_{n+1})$, where $x_1,\ldots,x_n \in nat$, and

$$(\forall i,j:1\leq i,j\leq n+1)\ x_i \equiv_E x_j \Rightarrow i=j,$$

so we need only consider the storage relations at t where t has the form eseq or $sq(x_1:\ldots:x_{n+1})$. Any condition holds for the empty relation $\Rightarrow_{eseq}^*$, so we only consider terms of sort *nes*. Before classifying the storage type we give some lemmas.

lemma 1: $lrem(sq(x_1:\ldots:x_{n+2})) \equiv sq(x_2:\ldots:x_{n+2})$ ♥

lemma 2: $rrem(sq(x_1:\ldots:x_{n+2})) \equiv sq(x_1:\ldots:x_{n+1})$ ♥

lemma 3: $\mathtt{left}(\mathtt{sq}(x_1:\ldots:x_{n+1})) \equiv x_1$ ♥

lemma 4: $\mathtt{right}(\mathtt{sq}(x_1:\ldots:x_{n+1})) \equiv x_{n+1}$ ♥

lemma 5: $\downarrow \mathtt{sq}(x_1:\ldots:x_{n+1})^* = \{[\mathtt{sq}(x_i:\ldots:x_j)] \mid 1 \le i \le j \le n+1\} \cup \{[\mathtt{eseq}]\}$ ♥

lemma 6: $\Downarrow \mathtt{sq}(x_1:\ldots:x_{n+1})^* = \{[x_1],\ldots,[x_{n+1}]\}$ ♥

lemma 7: $\rightarrow \mathtt{sq}(x_1:\ldots:x_{n+1})^* =$

$\{<[\mathtt{sq}(x_i:\ldots:x_j)],[\mathtt{sq}(x_i:\ldots:x_{j-1})]>, <[\mathtt{sq}(x_i:\ldots:x_j)],[\mathtt{sq}(x_{i+1}:\ldots:x_j)]>$

$\mid 1 \le i < j \le n+1\} \cup \{<[\mathtt{sq}(x_i)],[\mathtt{eseq}]> \mid, 1 \le i \le n+1\}$ ♥

lemma 8: $\Rightarrow \mathtt{sq}(x_1:\ldots:x_{n+1})^* =$

$\{<[x_i],[x_i]> \mid 1 \le i \le n+1\} \cup \{<[x_i],[x_{i+1}]> \mid 1 \le i < n+1\} \cup \{<[x_j],[x_{j-1}]> \mid 1 < j \le n+1\}$ ♥

Now we prove the storage type by induction.

lemma: $\Rightarrow_t^*$, where $t =_{def} \mathtt{sq}(x_1:\ldots:x_{n+1})$, is doubly-linked linear.

proof:

Base case: (n=0)

$t = x_1$

$\Downarrow t^* = \{[x_1]\}$

$\Rightarrow_t^* = \{\}$.

$\Rightarrow_t^*$ is trivially doubly-linked linear.

Induction step: Assume $n \in N$ and $\Rightarrow_t^*$, where $t =_{def} \mathtt{sq}(x_1:\ldots:x_{n+1})$, is doubly-linked linear.

Consider $s =_{def} \mathtt{sq}(x_1:\ldots:x_{n+2})$. By lemmas 1-8 we can show that

$\Downarrow s^* = \Downarrow t^* \cup \{[x_{n+2}]\}$ ♥

$\Rightarrow_s^* = \Rightarrow_t^* \cup \{<[x_{n+2}],[x_{n+2}]>\} \cup \{<[x_{n+2}],[x_{n+1}]>, <[x_{n+1}],[x_{n+2}]>\}$. ♥

When $\Rightarrow_t^*$ is (1:2)-regular and symmetric, all pairs in $(\Rightarrow_t^*)^*$ are comparable, and there are only two nodes with outdegree 1, then clearly

$$\Rightarrow_t^* \cup \{<[x_{n+2}], [x_{n+2}]>\} \cup \{<[x_{n+2}], [x_{n+1}]>, <[x_{n+1}], [x_{n+2}]>\}$$

also fulfills these conditions. $[x_{n+2}]$ and $[x_1]$ are the (labels of) nodes in $\Rightarrow_s^*$ with outdegree 1.

Conclusion: **Sequence** has doubly-linked linear storage type. □

Example 3.19: The **VNvariable** specification has the unstructured storage type.

Proof:
There are no rearranger or eliminators in this specification and `lookup` is the only selector. Therefore, for every t:`fsto`,

$\Downarrow t^*$ = {[`lookup(t)`]}

$\Rightarrow_t^*$ = {}.

3.4 Automating the ADT Classifications

Since the classification of an ADT by storage type will always require a proof by induction, this suggests that similar to the classification of an operator, inductionless induction proof methods might be appropriate. In order to apply these methods we must give an equational axiomatisation of storage relations; this is the topic of the next chapter.

When, for a given keyless/implicitly keyed ADT, we can give a specification of the *restricted* storage relations as a confluent and terminating rewriting system, and we can specify the properties given in §3.2.2 as rewrite rules, then we can classify a keyless/implicitly keyed specification by adding the specifications of properties, one by one, to the rewriting system and attempting to perform a proof by consistency. Some properties are easy to specify equationally; for example, the symmetry of a relation. Other properties, for example, upwards-directedness, require existential quantification. For these properties, we can either supply the witness (when we know it) or, when the contents sets are finite, (cf. §3.1.3), we can enumerate the elements and try each one as a witness.

There are problems; for example, we will, in general, be unable to orient the equations for $\Downarrow$ and $\Rightarrow$ in the desired direction using the standard (syntactic) orderings. For example, without going into too much detail, the equations for $\Downarrow$ and $\Rightarrow$ will be recursively defined and will involve equations of the form

$\Downarrow t = \ldots \Downarrow e(t) \ldots$

$\Rightarrow t = \ldots \Rightarrow e(t) \ldots$

where e is an eliminator.

If the eliminators may be regarded as the specifications of sub-term relations (as in the *BrW* classification), then the equations can be oriented from left to right Otherwise, they will be ordered in the reverse direction and new orderings based on the semantics of the keyed/implicitly keyed specification will be required. We shall return to this topic in chapter 4.

3.5 Summary

In this chapter we have shown how the classification of operators allows us to classify an arbitrary keyless or implicitly keyed specification. We have defined the class of storage relations of an ADT and described a method of classifying specifications by analysing the properties of some of those relations.

The storage relations are relations on the model of the specification and they reflect an "observational" view, with respect to the primitive sort, of how the data is stored, retrieved, and manipulated in elements of the ADT. The relations depend on the terms of an ADT and the definitions are given for any ADT: they refer only to the operator classification. The restricted storage relations of an ADT are a sub-class of the storage relations; they consist of those storage relations which depend on terms which fulfill certain syntactic criteria. ADTs are classified by the properties which hold for their restricted storage relations; several such properties are defined and some example specifications are classified according to them. The ADT classification may be automated using rewriting techniques, but this will require an equational axiomatisation of the storage relations and may also require the definition of new term orderings.

Chapter Four
Specifying Storage Relations

The definitions of contents sets and storage relations given in chapter 3 were given at the semantical level: they were defined in terms of the initial algebras which give meaning to a specification. In this chapter we consider how these sets and relations can be defined at the syntactical level: we consider how they can be specified equationally.

We first consider a formal approach in which the specifications of the storage relations are parameterised by the keyless/implicitly keyed ADTs. This approach is complex because in order to give the parameter requirement, a specification of the *meaning* of the keyless/implicitly keyed specifications is required. After outlining some of the problems involved with this approach, we adopt a less formal approach.

The second, hierarchical approach is less formal and consists of enriching the given keyless/implicitly keyed ADT with some additional sorts and equations. Only part of the enrichment is formally specified although we state some formal requirements which the enriched specification must satisfy.

4.1 A Parameterised Specification of Storage Relations

For a given keyless/implicitly keyed ADT (henceforth referred to as the *object* specification), the storage relations depend on two aspects of the object specification: the operator partition and the initial algebra which gives meaning to the object specification. A formal specification of the storage relations is a parameterised specification whose parameter requirement is some form of a term algebra with equivalences between terms; indeed, we intend that the *meaning* of the object specification is the actual parameter.

4.1.1 Specifications as Parameters

In this section we consider how the meaning of an object specification may be considered as the parameter in the specification of the storage relations. First, we give

the specification of the parameter requirement and then second, we give some (informal) rules for transforming a given object specification into one which meets the parameter requirement.

4.1.1.1 The Parameter Requirement

The parameter requirement must include the information about the classification of operators, the construction of terms, and the equivalences between terms induced by the equations of the object specification.

To specify the classification of operators, we associate a sort with each of the operator partition classes: the sorts *gen*, *elim*, *rear*, *sel* and *oth*. To specify the construction of terms, we include the sort *term* and an operator *app* (for "append") which constructs terms of sort *term* from terms of sort *gen*, *elim*, etc. and lists of sort *term*. Object sorts are represented by constants of sort *sort*. The construction of non well-typed terms (with respect to the object sorts) cannot be avoided and so two operators which specify the source and target arities of object operators (*sce* and *tgt* resp.) are included. Terms can be "filtered" (with the *bool*-sorted operator *ok*) to check that they are well-formed with respect to the source and target sorts of the object operator. In order to overload the operator *app*, the sort *op* is included as the super-sort of the sorts *gen*, *elim*, *rear*, *sel* and *oth*. The parameter requirement is given below as a meta theory in the style of CLEAR and OBJ2 [FGJ 85]:

```
meta Objspec =  enrich Bool by

data sorts   op, gen, elim, rear, sel, oth,
             term, terms,
             sort, word

subsorts     gen ≤ op, elim ≤ op, rear ≤ op, sel ≤ op, oth ≤ op

opns
app      : op terms -> term          !construct terms
sce      : op -> word                !op source sorts
tgt      : op -> sort                !op target sort
nils     : word
nilt     : terms
_:_      : term terms -> terms
_::_     : sort word -> word
δ        : sort                      !reps of object sorts
τ        : sort
ϕ        : sort
```

```
β         : sort
err       : sort
ok        : term -> bool              !check well-formed
mapsort   : terms -> word
match     : word word -> bool         !compare sortlists
sortof    : term -> sort
eqns
∀s:sel.          tgt(s)              = δ
∀e:elim.         tgt(e)              = τ
∀f:op,t:terms.   ok(app(f,t))        = match(sce(f),mapsort(t))
∀f:op,t:terms.   sortof(app(f,t))    = tgt(f)          if ok(app(f,t))
∀f:op,t:terms.   sortof(app(f,t))    = err             if ~(ok(app(f,t)))
∀t:term,l:terms.mapsort(t:l)         = sortof(t)::mapsort(l)
                 mapsort(nilt)       = nils
∀x,y:sort,l,l':word.
                 match(x:l,y:l')     = ((x=y)∨((x=τ)∧(y=φ)))∧match(l,l')
∀x:sort,l:list.  match(nils,x:l)     = F
∀x:sort,l:word.  match(x:l,nils)     = F
                 match(nils,nils)    = T
end
```

The parameter requirement does not include more requirements concerning the operators *sce* and *tgt* because many of the definitions of the operator classification given in §2.4.2 allow for either τ or φ in the description of the arities of the operators. We note that it would be possible to include in the parameter requirement the requirements of the operator classification. This would be similar to the classification of operators using inductionless induction described in §2.6.1. Here, we would have to specify multisets and the **leaves** operator, i.r. **leaves**: `term -> set`, as the multiset of terms whose sort is *term* and whose *tgt* is δ. The equations specifying the properties of rearrangers and eliminators would be included in the parameter requirement; for example, the following specification of the rearrangement property:

`∀r:rear,t:term.  ok(r,t) => (leaves(t) = leaves(app(r,t)))`

might be included. The operator partition requirements have not been included here in order to keep the parameter requirement as simple as possible.

Given the above parameter requirement **Objspec**, each object specification has to be transformed into one meeting these requirements; in the following section, we discuss such a transformation.

4.1.1.2 Transforming Specifications to meet the Parameter Requirement

By definition, a given object specification does not meet the requirements of **Objspec**; each object specification must be *represented* by another specification which meets the parameter requirement. In this section, we consider the construction of the representation.

Assuming that the object specification is **Spec0**, the representing specification is **Spec1**, and **Spec1** originally consists of the **Bool** specification; we consider how we can construct **Spec1** from **Spec0** by enrichment. Namely, we must enrich **Spec1** so that it meets the requirements of **Objspec** whilst ensuring that it represents the object specification **Spec0**. The definition of a correct representation (or implementation) of a specification is given in the following chapter, for the moment we assume an informal notion of representation. The enrichment is given by four steps and it includes adding (parts of) the body of **Objspec** to **Spec1**. Strictly speaking, **Objspec** is a metatheory; therefore we assume that **Req** is the specification whose body is exactly the same as that of **Objspec**. In addition, for brevity, we use the specification operation

forget sorts <sortlist> ***from*** <exp_2> ***end***

as an abbreviation for deriving, without renaming, all the sorts from the body of <exp_2> except those in <sortlist> and all the operators and equations from the body of <exp_2> except those which involve a sort from <sortlist>.

Before we give the steps, we give the specification of **Stack1**, the representation of the **Stack** specification, as an example. **Req** is not included in its entirety in **Stack1** because **Stack** does not include any rearranger operators; in order to avoid having an empty carrier (i.e. the carrier of *rear*) we hide the sort *rear*, and the operations which use it, in **Stack1**.

```
spec    Stack1 =
enrich  forget sorts rear
        from Req end
by
opns
0       : gen
succ    : gen
push    : gen
pop     : elim
top     : sel
empty   : oth
```

```
FF        : oth
TT        : oth
eqns
sce(top)          = ϕ::nils
sce(pop)          = τ::nils
sce(push)         = τ::(δ::nils)
sce(create)       = nils
sce(empty)        = τ:nils
tgt(empty)        = β
tgt(top)          = δ
tgt(pop)          = τ
tgt(push)         = ϕ
tgt(create)       = τ
sce(TT)           = nils
sce(FF)           = nils
tgt(TT)           = β
tgt(FF)           = β
∀s,d:term.
app(pop,(app(push,s:(d:nilt)):nilt))     = s
app(pop,(app(create,nilt):nilt))         = app(create,nilt)
∀s,d:term.
app(top,(app(push,s:(d:nilt)):nilt))     = d
∀s,d:term.
app(empty(app(push,s:(d:nilt)):nilt))    = app(FF,nilt)
app(empty,(app(create,nilt)):nilt))      = app(TT,nilt)
end
```

In general, we construct the specification of **Spec1**, from **Spec0** in the following four steps:

First, add the sorts given in **Req** to Σ_1 excepting that if the operator partition classes Σ_r or Σ_o are empty, then do not add the sorts *rear* or *oth* resp.

Second, add the operators given in **Req** to Σ_1, (excepting any operator which refers to a sort which has been excluded in the first step). Then, add the appropriate constants for the object operators, i.e. for each operator $f \in \Sigma_g$, add the constant `f:gen`, for each operator $f \in \Sigma_e$, add the constant `f:elim`, for each operator $f \in \Sigma_r$, add the constant `f:rear`, for each operator $f \in \Sigma_s$, add the constant `f:sel`, and for each operator $f \in \Sigma_o$, add the constant `f:oth`.

Third, add equations specifying the application of *sce* and *tgt* to the constants defined above. We should note that the sorts in Σ are represented by constants of sort

$sort$ in Σ_1; this can be a source of confusion because we need to refer to the *names* of those sorts at two levels. We shall assume that there is a representation mapping *srep* which maps the sorts in Σ to constants of sort $sort$ in Σ_1 in the obvious way: *srep*(τ)=τ, *srep*(ϕ)=ϕ, *srep*(δ)=δ, and *srep*(bool)=β (the sort name $bool$ is still used in Σ_1). For each constant (object) operator f of sort s in Σ, we add the equations

$$\mathtt{sce(f)} = \mathtt{nils}$$
$$\mathtt{tgt(f)} = \mathit{srep}(\mathtt{s})$$

to E_1. For each (object) operator f in Σ with arity $s_1 \ldots s_n \rightarrow s$ ($n \geq 1$), we add the equations

$$\mathtt{sce(f)} = \mathit{srep}\ (\mathtt{s}_1)::(\mathit{srep}\ (\mathtt{s}_2)::(\ldots(\mathit{srep}\ (\mathtt{s}_n)::\mathtt{nils})\ldots)))$$
$$\mathtt{tgt(f)} = \mathit{srep}(\mathtt{s}).$$

Fourth, add all the equations in **Req** and then add representations of the equations in **Spec0**. The representations are given by applying a representation mapping *rep* to both sides of the equations in **Spec0**. *rep* maps the sorts and terms (with S-sorted variables X) of **Spec0** to the sorts and terms (with S'-sorted variables X) of **Spec1** as follows:

$(\forall s \in S)$ $\quad \mathit{rep}:(T_\Sigma(X))_s \rightarrow (T_{\Sigma 1}(X))_{term}$

$(\forall s \in S)(\forall t \in \Sigma_{\lambda,s})$ $\quad \mathit{rep}(t) = \mathtt{app(t,nilt)}$

$(\forall x \in X)$ $\quad \mathit{rep}(x) = x.$

$(\forall n \geq 1)(\forall f \in \Sigma_{s1,\ldots,sn,s})(\forall w_1 \in (T_\Sigma(X))_{s1}, \ldots w_n \in (T_\Sigma(X))_{sn})$

$$\mathit{rep}(\mathtt{f}(w_1,\ldots,w_n)) = \mathtt{app}(\mathtt{f},(\mathit{rep}(w_1):(\mathit{rep}(w_2):(\ldots(\mathit{rep}(w_n)::\mathtt{nilt})\ldots)))).$$

We have not formally justified the claim that the representation we have described above is a *correct* representation of the object specification. In order to ensure that the representation has captured exactly the structure of the initial algebra of the object specification, we must formalise the relationship between the specifications constructed by the above four steps and the object specifications and show that the relationship is a correct *implementation* (c.f. chapter 5). Although this is possible, we do not pursue the topic as we shall soon encounter some problems which will force us to abandon this approach.

4.1.2 Axiomatising Relations

Given a specification which meets the requirements of **Objspec**, we now consider the specification of the contents sets and the storage relations. We note that although the specification of the contents sets is not strictly necessary, (i.e. the domain of a relation can be derived from the specification of the relation) the contents sets are included as they will be required later on in the implementation process.

One important consequence of specifying the contents sets and storage relation is the definition of equality (and inequality) on the (specified) set and relations. Equality between contents sets and storage relations was not explicitly discussed in chapter 3. In order to specify the contents sets and storage relations, we shall have to consider which notion of equality is appropriate. But first, we consider how to represent the contents sets and storage relations abstractly: there are two possibilities.

The first possibility is to specify $\Downarrow t$ and $\Rightarrow_t$ by "observing" their members; i.e. by boolean sorted operations. Because $\Downarrow t$ and $\Rightarrow_t$ depend on the derived sort, these relations are *bool*-sorted operations on terms (i.e. on the sort *term*). For example, the following parameterised specification **Storage_Relation** specifies $\Downarrow t$ and $\Rightarrow_t$ as binary and ternary relations on the sort *term*; as before, when the right hand side of a *bool*-sorted equation is T, it is omitted. (N.B the symbol => denotes implication and the symbol $\Rightarrow$ denotes the elimination relation)

```
proc    Storage_Relation(objspec:Objspec) =
enrich  objspec by

opns    _mem↓_      : term term -> bool
        _mem⇓_      : term term -> bool
        →           : term term term -> bool
        ⇒           : term term term -> bool

eqns
∀t:term.                t mem↓ t
∀t:term,r:rear.         t mem↓ app(r,t)
∀t,x:term,e:elim.       (x mem↓ t)   => app(e,x) mem↓ t
∀t,x:term,s:sel.        (x mem↓ t)   => app(s,x) mem⇓ t
∀t,x:term,e:elim.       (x mem↓ t)   => →(t,x,app(e,x))
∀t,x,y:term,s:sel. →(t,x,y)      => ⇒(t,app(s,x),app(s,y))

end
```

We should note that this specification specifies the relations on terms (of sort *term*) which do not represent terms (of sorts τ, ϕ, and δ) in the domains of $\downarrow t$, $\rightarrow_t$, and $\Rightarrow_t$ as given in chapter three. For example, in **SRelation(Stack1)**, (cf §4.1.1.2), we have

$$(\mathtt{app(0,app(0:nilt))}\ \mathtt{mem}\downarrow\ \mathtt{app(0,app(0,nilt))}) \equiv_E \mathtt{T}.$$

This equivalence is nonsensical because `app(0,app(0,nilt))` is neither a representation of a derived sort term, nor is it even well-formed with respect to the object sorts. However, in an implementation, equivalences between terms which are not representations of object terms are irrelevant; of course, they can be avoided using conditional equations. For example, the first two equations might be given as:

```
∀t:term.
(tgt(t)=τ ∨ tgt(t)=ϕ)                          => t mem↓ t
∀t:term,r:rear.
((tgt(t)=τ ∨ tgt(t)=ϕ) ∧ ok(app(r,t)))         => t mem↓ app(r,t)
```

Unfortunately, even when we consider only terms which are representations, this parameterised specification is not sufficiently-complete with respect to **Bool**; i.e. for a given t, we have not specified when terms (of sort *term*) are not members of $\Downarrow t$ and $\Rightarrow_t$. Thus, a third element is introduced into the carrier of *bool*. The problem arises from the use of existential quantification in the definitions of $\downarrow t$ and $\rightarrow_t$ given in §3.1.1. We can specify bounded quantifiers (i.e. quantification over a finite set) in the equational logic, but to do this we need to know, in advance, the number of operators in the eliminator and rearranging partitions. When we have this information, and hence know the number of constants of sort *elim* and *rear*, then we can specify $\rightarrow(t,x,y)$ by introducing an operation which enumerates over the eliminators $e_1, \ldots, e_n$ to check whether $y \equiv_E app(e_1, x)$ or $y \equiv_E app(e_2, x)$,..., or $y \equiv_E app(e_n, x)$. For example, in **SRelation(Stack1)**, there is only one constant of sort `elim` (i.e. `pop`) and so $\rightarrow t$ can be specified by :

$$\rightarrow(t,x,y) \quad = \quad (x\ \mathtt{mem}\downarrow t) \wedge (y = \mathtt{app(pop,x)})$$

Because we do not know (from the parameter requirement) the number of constants of sort *elim* and *rear*, a sufficiently-complete specification of the relations cannot be given using the current specification language. One solution to this problem is

to extend the specification language to include enumeration over operators and equation schemas.

Worse still, the definition of the set $\downarrow t$ does not use bounded quantification. For example, mem$\downarrow$ is specified by:

$x\ mem\downarrow t \quad = \exists w \in W.\ (w = x)$

where W is the set of terms defined inductively by:

1) $t \in W$,

2) $(\forall r{:}rear)\ \ app(r,t) \in W$,

3) $(\forall e{:}elim)\ (\forall w \in W)\ \ app(e,w) \in W$.

When the storage relation at t is finite, then a finite set can be substituted for W. However, the exact nature of the substitution depends on the the object specification.

Another way of specifying $\Downarrow t$ and $\Rightarrow_t$ is to introduce two new sorts and to explicitly construct $\Downarrow t$ and $\Rightarrow_t$ for each t:term; i.e. to construct "sets" of sort term and "sets" of ordered pairs of sort term. (We refer to these objects as "sets" although it is not yet obvious in what sense they are specifications of mathematical (finite) sets). We must also decide whether these objects should have an initial semantics (i.e. they are *data* sorts) or a loose semantics. Before we discuss this issue, we note that we again encounter the problem that we do not know in advance the number and names of the operators in the object specification. In order to specify $\Downarrow t$ and $\Rightarrow_t$, we must be able to quantify over the eliminators, rearrangers and selectors of the object specification, and to specify with equation *schemas*.

As an informal example, assume that the language includes equation schemas and we have the "pair" constructor <_,_>: term term -> pair, the "set" operators {}: set, _∪_: pair set -> set, and _U_: set set ->set. When we have exactly n constants, $e_1, \ldots, e_n$, of sort *elim*, and m constants, $r_1, \ldots, r_m$, of sort *rear*, then $\rightarrow t$ is (partially) specified by:

$\rightarrow t \quad = \ --{>}(t)\ U\ --{>}(app(r_1,t))\ U \ldots U\ --{>}(app(r_m,t))$

$--{>}t = \ (<t, app(e_1,t)> \cup \ldots \cup <t, app(e_n,t)>)$
$\qquad U\ --{>}(app(e_1,t))\ U \ldots U\ --{>}(app(e_n,t))$

We have omitted the quantification of t because this depends on the properties of the operators in object specification and the exact arity of each eliminator and rearranger. The specification is partial because $\downarrow t$, $\Downarrow t$, $\rightarrow_t$ and $\Rightarrow_t$ are specified by recursion equations; there is a problem of how to specify, in general, the base cases. For example, when the object specification contains a single eliminator e and for some term k, app(e,k) $\equiv_E$ k, then an equation such as $\downarrow k = k \cup \{\}$ should be included in the specification. We could make the "sets" absorptive; i.e. by add ing the equation $x \cup (x \cup S) = x \cup S$. Because of the partial ordering on the derived sort, this additional equation would make the specification of $\Downarrow t$ and $\Rightarrow_t$, the images of $\downarrow t$ and $\rightarrow_t$ under the selectors, more complex. We would have to ensure that selector constants (of sort *sel*) are only applied to terms (of sort *term*) which represent object terms from the domain of the object selector.

In conclusion, restricting the arities of the selectors, eliminators, and rearrangers would alleviate some of the problems with this approach, but not all. Unless we extend the specification language with constructs such as enumeration over operator symbols and equation schemas, then we cannot formally specify the storage relations as parameterised, equational specifications. Of course, the storage relations of any given object specification can be specified equationally, and we will discuss this further in the following section; our inability to give a *generic* specification only restricts the potential for automating our methodology. We hope to define suitable extensions to the *process* specification language (as opposed to the object specification language which is adequate for specifying object ADTs) in the future. Moreover, both approaches: specification by observation and specification by explicit construction, would benefit from the addition of higher-order operations into the specification language as described in [Möl 87].

We shall continue with the second kind of representation of $\Downarrow t$ and $\Rightarrow_t$ (i.e. by explicit construction), but the specifications will be hierarchical instead of parameterised. The "hierarchical" approach consists of associating with a given object specification **Spec**, a specification of the storage relations of **Spec**. This latter specification is usually called **SRelation** and it is an enrichment of **Spec**. The name of the specification of the storage relations does not depend on the name of the object specification thus indicating that there is no formal, general relationship between the two specifications; when we need to refer to a particular specification of storage relations we refer to the "the **SRelation** derived from **Spec**". Each **SRelation**

specification must meet some formal requirements as described in the following section.

4.2. Specifying SRelation

For a given object specification **Spec**, the **SRelation** derived from **Spec** consists of **Spec**, a suitable specification of "sets" of the primitive sort, "sets" of ordered pairs of the primitive sort, and a specification of the operators $\Downarrow$ and $\Rightarrow$. Because $\Rightarrow$t may be regarded as a directed graph, we refer to the "sets" of the primitive sort as *nodes*, and the "sets" of ordered pairs of the primitive sort, as *edges*.To summarise informally,

SRelation = Spec + Nodes_and_Edges + $\Downarrow$_and_$\Rightarrow$.

Only the second component is formally specified although some formal requirements for the third component and the entire specification are given. Before discussing these requirements, we must first clarify the issue of equality on storage relations.

4.2.1 Equality on Storage Relations

The issue which concerns us here is how should equality on the derived sort relate (if at all) to equality on the storage relations and contents sets. The equality we refer to is the congruence relation $\equiv_E$ which is generated by the specification of **SRelation**. It is obvious from the definitions in chapter 3 that we require

(P1) $(\forall s, t:\tau)$ $(([t] = [s]) \Rightarrow([\Downarrow t]= [\Downarrow s])) \wedge (([t] = [s]) \Rightarrow ([\Rightarrow t]= [\Rightarrow s]))$.

This proposition is of course true by the definition of the congruence $\equiv_E$. The question is should we allow $[\Downarrow t] = [\Downarrow s]$ or $[\Rightarrow t] = [\Rightarrow s]$ when $[t] \neq [s]$? In essence, we need to decide on the properties of the "sets" which are the values of $\Downarrow$t and $\Rightarrow$t; namely, should a theory of finite sets be included in the specification. To illustrate the discussion, assume that we have a specification of *col* (for collection) and the operator $\Rightarrow$ has arity τ -> col.

If the specification of `col` is anarchic (i.e. there are no equations), then in the **Reversible_List_1** example, we would have

(P2) $(\forall t,s:\mathtt{nel})\ ([t]=[s]) \Leftrightarrow ([\Downarrow t]=[\Downarrow s]) \Leftrightarrow ([\Rightarrow t]=[\Rightarrow s])$.

If, on the other hand, the specification of *col* has an associative, commutative generator as is usual for specifications of finite sets (for example, in the specification of FINSET in [Bro 84]), then **(P2)** may no longer hold. For example, **(P2)** (with `t,s` of sort `nes` and `neq` resp.) still holds in the **Stack** and **Queue** examples, but it no longer holds in the **Reversible_List_1** example; i.e. we now have

(P3) $(\exists l:\mathtt{nel})\ ([l] \neq [\mathtt{rev}(l)]) \wedge ([\Rightarrow l]=[\Rightarrow \mathtt{rev}(l)])$.

The motivation for specifying storage relations is to use them as *representations* of the derived sort elements of the object ADTs. Representations are discussed formally in the following chapter. Informally, one expects that a representation should not identify elements which are not already identified in the abstraction; i.e. **(P3)** is undesirable because as a representation, $\Rightarrow$ identifies `l` and `rev(l)`. To be more precise, the identification of `l` and `rev(l)` is undesirable because it implies the identification of the primitive terms `hd(l)` and `hd(rev(l))`. Storage relations reflect an observational view of the primitive sort; clearly the theory of the primitive specification should not be altered in any way.

However, there is no reason why elements of the derived sort should not be identified in the representation; provided that the identification does not imply new equivalances between elements of the primitive sort. For example, if we consider the **Sequence** specification without the associativity axiom (E13), then we can find `seq` sorted terms `l` and `l'` which are not equivalent (in either the equational or inductive theory) but which are indistinguishable w.r.t. the primitive sort. Namely, there is no finite sequence of selectors s and eliminators $e_1,..,e_n$ such that

$$[s(e_n(\ldots e_1(l)\ldots))] \neq [s(e_n(\ldots e_1(l')\ldots))]$$

i.e. there is a model in the class of loose models in which `l` and `l'` are identified. It therefore seems quite reasonable that we may have $[\Rightarrow l]=[\Rightarrow l']$.

4.2.2 Specifying Nodes_and_Edges

The second component of **SRelation**, **Nodes_and_Edges**, is specified by the parameterised specification **Nodes&Edges**.It is intended that the primitive specification (of the object specification) is the actual parameter; for example, **Nodes&Edges(Nat)**. The specifications **Col** and **Pair** are local to **Nodes&Edges**. The motivation for the subsort *rgraph* of *graph* is to enable us to distinguish between pairs of nodes and edges which are representations from those which are not.

```
meta U =   enrich Bool by data sorts elem   end

spec    Col(D:U) =  enrich D by

data sorts col

gen
[]        : col
_+1_      : elem col -> col          /*left insertion*/
end

spec    Pair (D:U) =  enrich D by

data sorts pair

gen
<_,_>     : elem elem -> pair
end

spec    Nodes&Edges(D:U) =
enrich
        derive   sorts ncol
              gen []n : ncol,  +1n: elem ncol -> ncol
        from Col(D) by ncol is col, []n is [], +1n is +1
        +
        derive   sorts ecol
              gen []e : ncol,  +1e: elem ncol -> ncol
        from Col(Pair(D)) [elem is pair]
              by ecol is col,[]e is [], +1e is +1
by
data sorts graph

subsorts rgraph ≤ graph

gen
[_,_]     : ncol ecol -> graph
end
```

4.2.3 Requirements of SRelation

Now we are able to formalise the requirements of **SRelation**.

When **Spec** is the object ADT over the primitive specification **Prim**, then the **SRelation** derived from **Spec** includes both **Spec** and **Nodes&Edges(Prim)**. The third component of **SRelation**, ⇓_and_⇒, cannot be specified because of the problems described in §4.1.2; instead, we give the requirements which any specification of **SRelation** must fulfill. The requirements refer to the sorts τ and ϕ from the object specification and they are given by:

i) the signature of **SRelation** contains the operators

⇓ : τ -> ncol

⇒ : τ -> ecol

rep : τ -> rgraph

abs : rgraph -> τ.

ii) **SRelation** is sufficiently-complete and consistent with respect to **Prim** and the equations for ⇓, ⇒, abs and rep are well-spanned.

iii) The equations of **SRelation** include

$(\forall t{:}\tau)$ rep(t) = ([⇓t, ⇒t]).

$(\forall t{:}\tau)$ abs([⇓t, ⇒t]) = t.

We note that **(P2)**, $(\forall t, s{:}\tau)\ (t \equiv_E s) \Leftrightarrow (\Downarrow t \equiv_E \Downarrow s) \Leftrightarrow (\Rightarrow t \equiv_E \Rightarrow s)$, is implied by conditions ii) and iii).

The requirement for **SRelation** allows for the possibility that for some object specifications, several specifications, with non-isomorphic models, may fulfill the requirements. This is because the requirement does not place any obligations on the terms of sort *ncol* and *ecol* which are not equivalent to a contents set or storage relation resp. That is, it does not oblige equivalences, nor inequivalences, between elements of *ncol* and *ecol* which are not *representations*. Namely, the equations in the third component of **SRelation** (i.e. in the ⇓_and_⇒ component) may be more than sufficient to satisfy **(P2)**; they can induce equivalences between terms of sort *ncol* and

ecol which make more elements of these sorts into representations. Example 4.3 in the following section illustrates this situation.

4.2.4 Overspecification and Enrichment

If **SRelation** is sufficiently-complete and consistent w.r.t. **Prim**, but not to **Spec + Nodes&Edges(Prim)**, then the specification of SRelation preserves the loose semantics of **Spec** but not the initial semantics of **Spec**. In the following chapter we will discuss how **Spec** is *implemented* by its corresponding **SRelation**. If **SRelation** is sufficiently-complete and consistent w.r.t. **Spec**, then we can easily synthesise an implementation of **Spec** by **SRelation**, w.r.t. to the initial semantics of **Spec**. Instead of liberalising the notion of implementation as suggested in [BMP 86] to explicitly include loose semantics, we shall enrich the object specification **Spec** to make it *overspecified* (cf. §2.3.2). Thus, we can keep one simple notion of implementation.

In conclusion, when **SRelation** is not sufficiently-complete and consistent w.r.t. **Spec + Nodes&Edges(Prim)**, then we enrich **Spec** with equations **E** so that **SRelation** is sufficiently-complete and consistent w.r.t. **Spec + E + Nodes&Edges(Prim)**. **Spec + E** is then an overspecified specification; for example the Sequence specification with E3 is overspecified. In subsequent chapters, we assume that object specifications are overspecified.

4.3 Examples of SRelation

We conclude this chapter with some examples of **SRelation** .

Example 4.1: Stack

Spec **SRelation1 =** ***enrich*** **Stack + Nodes&Edges(Nat)** ***by***

opns

```
⇒       : stack -> ecol
⇓       : stack -> ncol
rep     : stack -> rgraph
abs     : rgraph -> stack
```

eqns

∀s:stack.	rep(s)	= [⇓s,⇒s]	S1
∀s:nes.	⇓(s)	= top(s) $+1_n$ ⇓(pop(s))	S2
	⇓(create)	= $[]_n$	S3
∀s:nes,n:nat.	⇒(push(s,n))	= <n,top(s)> $+1_e$ ⇒s	S4
∀n:nat.	⇒(push(create,n))	= $[]_e$	S5
	⇒(create)	= $[]_e$	S6
∀s:stack.	abs([⇓s,⇒s])	= s	S7

end

Example 4.2: Queue

Spec **SRelation2 =** ***enrich*** **Queue + Nodes&Edges(Nat)** ***by***

opns

```
⇒      : queue -> ecol
⇓      : queue -> ncol
rep    : queue -> rgraph
abs    : rgraph -> queue
```

eqns

∀q:queue.	rep(q)	= [⇓q,⇒q]	Q1
∀q:fullq.	⇓(q)	= front(q) $+1_n$ ⇓dequeue(q)	Q2
	⇓(eq)	= $[]_n$	Q3
∀q:fullq,n:nat.			
⇒(add(q,n)) =	<front(add(q,n),front(dequeue(add(q,n))> $+1_e$ ⇒dequeue(add(q,n))		Q4
∀n:nat.	⇒(add(eq,n)) =	$[]_e$	Q5
	⇒(eq)	= $[]_e$	Q6
∀q:queue.	rep(q)	= [⇓q,⇒q]	Q7
∀q:queue.	abs([⇓q,⇒q])	= q	Q8

end

Example 4.3: Queue

Spec **SRelation3 =** ***enrich*** **SRelation2** ***by***

eqns

∀e,e':nat. $n +1_e (n' +1_e []_e) = n' +1_e (n +1_e []_e)$ Q9

end

This specification is also derived from **Queue** and fulfills the requirements; but because of the additional axiom, Q9, (its model) is not isomorphic to (the model of) the specification in Example 4.2. More terms of sort `ecol` are representations in this specification than in Example 4.2. For example, in the previous specification, the term

$$\langle 1,2\rangle +1_e \ (\langle 2,3\rangle +1_e \ []_e)$$

is equivalent to

$$\Rightarrow(\mathrm{add}(\mathrm{add}(\mathrm{add}(eq,1),2),3)),$$

but there is no queue term `q` such that

$$\langle 2,3\rangle +1_e \ (\langle 1,2\rangle +1_e \ []_e)$$

is equivalent to $\Rightarrow q$; i.e. $\langle 2,3\rangle +1_e \ (\langle 1,2\rangle +1_e \ []_e)$ is not a representation. In the specification of Example 4.3,

$$\langle 2,3\rangle +1_e \ (\langle 1,2\rangle +1_e \ []_e)$$

is equivalent to

$$\langle 1,2\rangle +1_e \ (\langle 2,3\rangle +1_e \ []_e)$$

(because of Q9) and so

$$(\langle 2,3\rangle +1_e \ (\langle 1,2\rangle +1_e \ []_e)$$

is a representation.

Example 4.4: Binary_Tree_2

Spec **SRelation4 = *enrich*** **Binary_Tree_2 + Nodes&Edges(Nat)** ***by***

opns

```
⇒          : tree -> ecol
⇓          : tree -> ncol
rep        : tree -> rgraph
abs        : rgraph -> tree
plus_n     : ncol ncol -> ncol
plus_e     : ecol ecol -> ecol
```

eqns

```
∀t:tree. rep(t)    = [⇓t,⇒t]                                          T1

∀t:nonleaf. ⇓(t)   = root(t) +l_n (⇓left(t) plus_n ⇓right(t))         T2
∀n:nat.  ⇓(m(n))   = n +l_n []_n                                      T3

∀t:nonleaf.
         ⇒(t)      = (<root(t),root(left(t)> +l_e
                     (<root(t),root(right(t)> +l_e []_e))
                     plus_e (⇒left(t) +l_e (⇒right(t) +l_e []_e))     T4
∀n:nat.  ⇒(m(n))   = []_e                                             T5

∀N,N':ncol,n:nat.n +l_n (N plus_n N')  = (N plus_n (n +l_n N'))       T6
∀E,E'ecol,e:pair.e +l_e (E plus_e E')  = (E plus_e (e +l_e E'))       T7
∀N:ncol. []_n plus_n N       = N                                      T8
∀E:ecol. []_e plus_e E       = E                                      T9

∀t:tree. abs([⇓t,⇒t])        = t                                      T10
```

end

Example 4.5: Reversible_List_1

Spec **SRelation5 = *enrich*** **Reversible_List_1 +**
Nodes&Edges(ANat) ***by***

opns

```
⇒          : list -> ecol
⇓          : list -> ncol
rep        : list -> rgraph
abs        : rgraph -> list
plus_n     : ncol ncol -> ncol
plus_e     : ecol ecol -> ecol
down       : list -> ncol
rdown      : list -> ecol
```

eqns

```
∀l:list. rep(l)       = [⇓l,⇒l]                                          L1

∀l:nel.  ⇓(l)         = down(l) plus_n down(rev(l))                      L2
         ⇓(nil)       = []_n                                             L3

∀l:nel.  down(l)      = hd(l) +l_n down(tl(l))                           L4
         down(nil)    = []_n                                             L5

∀l:nel.  ⇒(l)         = rdown(l) plus_e rdown(rev(l))                    L6
         ⇒(nil)       = []_e                                             L7

∀n:nat,l:nel.
         rdown(n:l) = <hd(n:l),hd(tl(n:l))> +l_e(rdown(tl(n:l))         L8
∀n:nat.  rdown(n:nil)      = []_e                                         L9
       —rdown(nil) = []_e                                               L10

∀N,N':ncol,n:nat.n +l_n (N plus_n N')  = (N plus_n (n +l_n N'))          L11
∀E,E':ecol,e:pair.e +l_e (E plus_e E') = (E plus_e (e +l_e E'))          L12

∀N:ncol. []_n plus_n N       = N                                         L13
∀E:ecol. []_e plus_e E       = E                                         L14

∀l:list. abs([⇓l,⇒l])        = l                                         L15
```

end

Example 4.6: Circular_List_right

Spec **SRelation6 = *enrich*** **Circular_List_right +**
Nodes&Edges(Nat) *by*

opns

```
⇒        : list -> ecol
⇓        : list -> ncol
rep      : list -> rgraph
abs      : rgraph -> list
plus_n   : ncol ncol -> ncol
plus_e   : ecol ecol -> ecol
down     : list -> ncol
rdown    : list -> ecol
```

eqns

```
∀l:list. rep(l)        = [⇓l,⇒l]                                          L1

∀l:nel.  ⇓(l)          = down(l) plus_n down(shift(l))                    L2
         ⇓(nil)        = []_n                                             L3

∀l:nel.  down(l)       = hd(l) +l_n down(tl(l))                           L4
         down(nil)     = []_n                                             L5

∀l:nel.  ⇒(l)          = rdown(l) plus_e rdown(shift(l))                  L6
         ⇒(nil)        = []_e                                             L7

∀n:nat,l:nel.
         rdown(n:l) = <hd(n:l),hd(tl(n:l))> +l_e rdown(tl(n:l))           L8
∀n:nat.  rdown(n:nil)      = []_e                                         L9
         rdown(nil) = []_e                                                L10

∀N,N':ncol,n:nat.n +l_n (N plus_n N')  = (N plus_n (n +l_n N'))           L11
∀E,E':ecol,e:pair.e +l_e (E plus_e E') = (E plus_e (e +l_e E'))           L12

∀N:ncol. []_n plus_n N       = N                                          L13
∀E:ecol. []_e plus_e E       = E                                          L14

∀l:list. abs([⇓l,⇒l])        = l                                          L15
```

end

Example 4.7: Circular_List_left

Spec **SRelation7 = *enrich* Circular_List_left + Nodes&Edges(Nat) *by***

opns

```
⇒        : list -> ecol
⇓        : list -> ncol
rep      : list -> rgraph
abs      : rgraph -> list
plus_n   : ncol ncol -> ncol
plus_e   : ecol ecol -> ecol
down     : list -> ncol
rdown    : list -> ecol
```

eqns

$\forall$l:list. rep(l) = [⇓l,⇒l] L1

$\forall$l:nel. ⇓(l) = down(l) $plus_n$ down(shift(l)) L2

⇓(nil) = $[]_n$ L3

$\forall$l:nel. down(l) = hd(l) $+l_n$ down(tl(l)) L4

down(nil) = $[]_n$ L5

$\forall$l:nel. ⇒(l) = rdown(l) $plus_e$ rdown(shift(l)) L6

⇒(nil) = $[]_e$ L7

$\forall$ n:nat,l:nel.

rdown(n:l) = <hd(n:l),hd(tl(n:l))> $+l_e$ rdown(tl(n:l)) L8

$\forall$n:nat. rdown(n:nil) = $[]_e$ L9

rdown(nil) = $[]_e$ L10

$\forall$N,N':ncol,n:nat.n $+l_n$ (N $plus_n$ N') = (N $plus_n$ (n $+l_n$ N')) L11

$\forall$E,E':ecol,e:pair.e $+l_e$ (E $plus_e$ E') = (E $plus_e$ (e $+l_e$ E')) L12

$\forall$N:ncol. $[]_n$ $plus_n$ N = N L13

$\forall$E:ecol. $[]_e$ $plus_e$ E = E L14

$\forall$l:list. abs([⇓l,⇒l]) = l L15

end

4.3.1 The Correctness of SRelation

In this section we discuss the issues of proving the correctness of an **SRelation** specification using **SRelation1**, the **SRelation** derived from **Stack**, as an example.

SRelation1 must meet the requirements given in §4.2.3.; of course, in addition, **SRelation1** should also be correct with respect to the semantic definitions given in chapter 3.

The latter aspect can be established by defining a relationship between the classes in the model of **SRelation1** and the classes, from the model of **Stack**, in the contents sets and storage relations. For example, to show that the contents sets are correctly specified, we would have to show that

$(\forall t \in (T_\Sigma)_{nes})\ (\forall C \in \Downarrow t)\ (\forall s \in C)$

$(\exists x \in [\Downarrow t])\ (\exists n \geq 1)\ ((x = s_1\ +1_n\ s_2\ +1_n \ldots\ s_n\ +1_n\ []_n) \wedge ((\exists i{:}1 \leq i \leq n)\ s_i \equiv s))$

where $[\Downarrow t]$ is a class in the model of **SRelation1**,

$\Downarrow t$ is a set of classes of terms from the model of **Stack**,

$\equiv$ is the congruence generated by the equations of **Stack**.

Here, we concentrate on establishing the former aspect of correctness: **SRelation1** should satisfy the requirements given in §4.2.3. **SRelation1** trivially satisfies conditions i) and iii); it remains to show that **SRelation1** is sufficiently-complete and consistent w.r.t **Stack + Nodes&Edges(Nat)**.

By the definitions of §2.3.1 and [EhM 85], **SRelation1**$=(\Sigma,E)$ is sufficiently-complete and consistent w.r.t **Stack**$=(\Sigma_s,E_s)$ when the model of **Stack** is isomorphic to the Σ_s-reduct of the model of **SRelation1**; i.e. when the homomorphism $h : T_{\Sigma s, E s} \rightarrow (T_{\Sigma,1})_{\Sigma s}$ is surjective (for completeness) and injective (for consistency). When **SRelation1** is sufficiently-complete and consistent w.r.t **Stack**, then using the terminology of [Ehr 85], we say that **SRelation1** is *persistent* w.r.t **Stack**.

These "semantic" criteria are difficult to prove. Under certain circumstances, we can apply the results of [Pad 85] which define the equivalent "syntactic" criteria: properties of the associated term rewriting systems.

The approach of [Pad 85] divides a specification (Σ,E) into two subspecifications: the "base" specification (Σ_b,E_b) and the "parameter" specification (Σ_p,E_p). Informally, the division must satisfy the conditions that the base sorts are the specification sorts, the parameter specification is included in the base specification, the parameter specification includes the usual specification of Bool, and its model must give discrete interpretations to T and F.

We summarise the definitions and notations from [Pad 85] which we refer to in the following way. $=>_E$ denotes the (usual) *simple reduction relation* generated by E. E is *linear* if for each l=r in E each variable occurs at most once in l. $=>_{=E}$ denotes the (reflexive closure of the) *parallel reduction relation* generated by E. This relation combines independent, simple reductions into one reduction step. A term $t \in T_\Sigma$ has a E-*normal iorm* if $(\exists t' \in T_{\Sigma b})\ t =>_E t'$. E is *normalising* if $(\forall t \in T_\Sigma)$ t has an E-normal form. The definition of a *critical pair* is standard. A *parallel critical pair* results from the situation when several equations apply during parallel reduction; a *recursive critical pair* results from the more complicated situation where a subterm of a left hand side of an equations is also the prefix of another left hand side. A critical pair $<c_1,c_2>$ is E-*convergent* if

$$(\exists t,t' \in T_{\Sigma b})\ ((c_1 =>_E t) \wedge (c_2 =>_E t') \wedge (t =>_{=Eb} t')).$$

The main results of [Pad 85] are the Persistency Theorems I and II which we state below using our notation and without conditional equations. $\Sigma\backslash\Sigma'$ denotes the signature consisting of Σ without the intersection of Σ and Σ'; E\E' denotes the set of equations consisting of E without the intersection of E and E'.

Corollary 2.13 (Persistency Theorem I) from [Pad 85]:

(Σ,E) is persistent w.r.t. (Σ_p,E_p) if for all $\sigma \in \Sigma_{w,s}$, $s \in S_p$ implies $\sigma \in \Sigma_p$ and for all equations l=r in E, $sort(l) \in S_p$ implies $l=r \in E_p$.

Persistency Theorem II from [Pad 85]:

$(S,\Sigma E)$ is persistent w.r.t. (S_p,Σ_p,E_p) if

i) for all $\sigma \in \Sigma_b$, $sort(\sigma) \in S_p$ implies $\sigma \in \Sigma_p$,

ii) for all l=r in E_b, $variables(r) \subseteq variables(l)$ and $sort(l) \in S_p$ implies $l=r \in E_p$,

iii) for all l=r in $E \backslash E_b$, l contains at least one operation symbol from $\Sigma \backslash \Sigma_b$, but no operation symbols from $\Sigma_p \backslash \{T,F\}$,

iv) $E \backslash E_b$ is linear, terminating, and normalising,

v) all critical pairs of $E \backslash E_b$ into $E \backslash E_b$, all critical parallel pairs of E_b into $E \backslash E_b$ and all recursive critical pairs of $E \backslash E_b$ into E_b are $(E \backslash E_b)$-convergent.

A proof of persistency of an **SRelation** specification using the above results has to be organised into a hierarchy of proofs. In our example, we first show that **Nat** is persistent w.r.t. **Bool**, then we regard **Nat** as the parameter, **StackB** (**Stack** without operators *top* and *empty*) as the base, and show that **Stack** is persistent w.r.t **StackB**. Second, we regard **Stack** as the parameter, **Stack + Nodes&Edges(Nat)** as the base, and show that **SRelation1** is persistent w.r.t **Stack**.

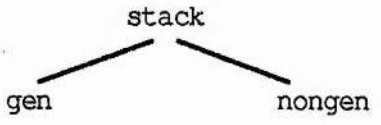
There may be some problems organising the given specification into a form of term rewriting system which allows the Persistency Theorem to be applied. Namely, the requirements of both termination and condition iii) may be difficult to fulfill. For example, certainly equations S1 and S3 in **SRelation1** can be oriented from left to right using the recursive path ordering [Der 85], but unless we force innermost rewriting, equation S2, when oriented from left to right, leads to an infinite sequence of rewrites. However, because the elimination relation coincides with the sub-term relation in this example, we could give S2 as:

$\forall s{:}stack.\ \Downarrow(push(s,n)) = n +1_n \Downarrow s$ S'2

which is easily ordered from left to right.

Unfortuately, this technique is not applicable to many other specifications in which the elimination relation is not the subterm relation. For example, Q2 in the **SRelation2** is not orientable from left to right with the Knuth-Bendix ordering nor with the recursive path ordering; nor can it be rewritten into a rule which can be oriented from left to right using those orderings. One solution, (suggested by A.J.J.Dick), is to define a new kind of termination ordering which is based on the properties of eliminators and the inverse of the partial ordering on the sorts. The (sort) ordering reflects the increasing size of terms; the termination ordering would capture the intuition of elimination and the fact that the arguments to $\Downarrow$ and $\Rightarrow$ are decreasing in size; i.e. on the left hand side of S2 $\Downarrow$ is applied to a term of sort `nes` whereas on the right hand side, $\Downarrow$ is applied to a term of sort `stack` (because `pop: stack -> stack`).

Moreover, we might also be able to use the subsorting to simulate innermost rewriting by partitioning terms into generator, or irreducible terms, and non-generator, or reducible, terms. By introducing the subsorts `gen` and `nongen` into `stack`:



and giving the arities of `create`, `push` and `pop` by

```
create  :  gen
push    :  gen nat -> gen
push    :  nongen nat -> nongen
pop     :  nongen -> nongen
pop     :  gen -> nongen
```

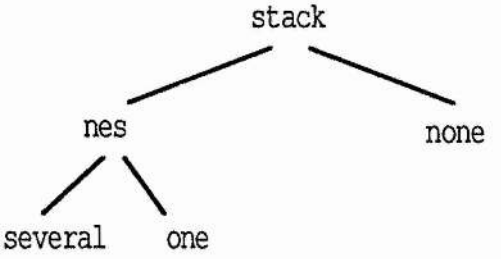
then because the sorts `gen` and `nongen` are incomparable, the equation

$$\forall g{:}gen, n{:}nat.\quad \Downarrow(push(g,n)) = top(push(g,n)) +1_n \Downarrow(pop(push(g,n)))$$

can be oriented from left to right.

The second problem, restricting the occurrence of the parameter operators on the left hand side of rules, can also be solved with partially-ordered sorts. When we regard **Stack** as the parameter, then condition iii) of Persistency Theorem II demands that the operators `push` and `create` do not occur in the left hand sides. We could meet this requirement by introducing subsorts; for example, by introducing the subsorts `none`,

one and several:



and giving the arities of `create`, `push` and `pop` by

```
create  : none
push    : one nat -> several
push    : none nat -> one
push    : several nat -> several
pop     : none -> none
pop     : one -> none
pop     : several -> nes
```

Now we can give equations S4,S5 and S6 in a form which meets condition iii):

$\forall s{:}\mathtt{several}.\quad \Rightarrow(s) = \langle top(s), top(pop(s))\rangle +1_e \Rightarrow(pop(s))$ S'4

$\forall s{:}\mathtt{one}.\quad \Rightarrow(s) = []_e$ S'5

$\forall s{:}\mathtt{none}.\quad \Rightarrow(s) = []_e$ S'6

In conclusion, the termination of some of the examples remains an open problem. Concerning our example, we show in the following two subsection, using the Persistency Theorems I and II, that **`Stack`** is persistent w.r.t **`Nat`** and if **`SRelation1`** is terminating, then **`SRelation1`** is persistent w.r.t **`Stack`**.

4.3.1.1 Persistency of `Stack`

In the following, we adopt the notation:

$(\Sigma_b,E_b) =_{def}$ **`Bool`**,

$(\Sigma_n,E_n) =_{def}$ **`Nat`**,

$(\Sigma_{s'},E_{s'}) =_{def}$ **`StackB`** $=_{def} (\Sigma_s\backslash\{top,empty\},E_s\backslash\{R1,R4,R5\})$.

$(\Sigma_s, E_s) =_{def}$ **Stack**

Lemma 1

Nat is persistent w.r.t **Bool**.

proof: by Corollary 2.13

Every equation of sort *bool* in E_n is in E_b. □

Lemma 2

StackB is persistent w.r.t **Nat**.

proof: by Corollary 2.13

Every equation of sort *nat* or *bool* in $E_{s'}$ is in E_n. □

Proposition 1: for all $s \in (T_{\Sigma s})_{stack}$, `empty(s)` has an E_s normal form.

proof: for all $s \in (T_{\Sigma s})_{stack}$, either `s=create`, or `s=push(t,x)`, with $t \in (T_{\Sigma s})_{stack}$, and $x \in (T_{\Sigma s})_{nat}$. If `s=create`, then `empty(s)` $=>_{Es}$ `T` (by R5); otherwise, $s \in (T_{\Sigma s})_{nes}$ (by the arity of `push`), and `empty(s)` $=>_{Es}$ `F` (by R6). Obviously, `T,F` $\in (T_{\Sigma s'})$.

Proposition 2: for all $t \in (T_{\Sigma s})_{nat}$, `t` has an E_s normal form.

proof: Every term `t` reduces to a term $t' \in (T_{\Sigma s'})_{nat}$ by induction on the number of occurrences of `push` in `t` and the reduction `top(push(s,x))` $=>_{Es}$ `x` (by R1).

Lemma 3

Let **Stack** be the parameterised specification, **StackB** the base specification and **Nat** the parameter specification. **Stack** is persistent w.r.t **Nat**.

proof: by Persistency Theorem II

i) There are no *nat* or *bool* sorted operations in $\Sigma_{s'} \setminus \Sigma_n$.

ii) There are no *nat* or *bool* sorted equations in $E_{s'} \setminus E_n$.

iii) All the l.h.s.'s of equations in $E_s \setminus E_{s'}$ contain at least one symbol from {*top, empty*} and no symbols from {*0, succ*}.

iv) $E_s \setminus E_{s'}$ is linear, terminating (using KB-ordering) and E_s-normalising because by

Propositions 1 and 2, for all $s \in (T_{\Sigma s})_{stack}$, $empty(s)$ has an E_s-normal form and for all $s \in (T_{\Sigma s})_{nes}$, $top(s)$ has an E_s-normal form.

v) Using the definitions and the fact that the outermost symbol of every l.h.s of $(E_s \setminus E_{s'})$ is a symbol from $(\Sigma_s \setminus \Sigma_{s'})$, we can show, in a manner similar to Example 5.13 in [Pad 85], that there are no critical pairs of $(E_s \setminus E_{s'})$ into $(E_s \setminus E_{s'})$, no parallel critical pairs of $E_{s'}$ into $(E_s \setminus E_{s'})$ nor any recursive critical pairs of $(E_s \setminus E_{s'})$ into $E_{s'}$.

□

4.3.1.2 Persistency of SRelation1

In this section we use (Σ_B, E_B) to denote **Stack + Nodes&Edges (Nat)** and (Σ, E) to denote **SRelation1** with S'4, S'5 and S'6 (as given above) substituted for S4, S5, and S6 resp.

Lemma 4

Stack + Nodes&Edges (Nat) is persistent w.r.t **Stack**.

proof: by Corollary 2.13

Every equation in E_B is in E_s. □

Proposition 3: for all $t \in (T_{\Sigma s})_{stack}$, $\Downarrow t$ has an E-normal form.

proof: by induction.

Base Case: When t has no occurrences of $push$ then t has a E_s-normal form because $\Downarrow t \Rightarrow_E []_n$ (by S3).

Induction Step: Assume $n \in N$, t contains $n \geq 0$ occurrences of $push$ and $\Downarrow t$ has E-normal form t'. Consider $push(t, x)$ for some $x \in (T_\Sigma)_{nat}$.

$\Downarrow push(t,x) \Rightarrow_E top(t) +1_n \Downarrow pop(push(t,x))$ (by S2)

$\Rightarrow_E x +1_n \Downarrow pop(push(t,x))$ (by R1)

$\Rightarrow_E x +1_n \Downarrow t$ (by R2)

$\Rightarrow_E x +1_n t'$ (by ass.)

By definition of Σ and Σ_B, every term in $(T_\Sigma)_{nat}$ is also a term in $(T_{\Sigma B})_{nat}$; therefore x is a E-normal form and $x +1_n t'$ is a E-normal form.

Proposition 4: for all $t \in (T_{\Sigma s})_{stack}$, $\Rightarrow t$ has an E-normal form.

proof: by induction.

Base Case: When t has no occurrences of $push$ then t has a E_s-normal form because $\Rightarrow t \Rightarrow_E []_n$ (by S6).

Induction Step: Assume $n \in N$, t contains $n \geq 0$ occurrences of $push$ and $\Rightarrow t$ has E-normal form t'. Consider $push(t,x)$ for some $x \in (T_\Sigma)_{nat}$.

$\Rightarrow push(t,x) \Rightarrow_E \langle x,top(t)\rangle +1_e \Rightarrow pop(push(t,x))$ (by S2)

$\Rightarrow_E \langle x,top(t)\rangle +1_e \Rightarrow t$ (by R2)

$\Rightarrow_E \langle x,top(t)\rangle +1_e t'$ (by ass.)

By definition of Σ and Σ_B, every term in $(T_\Sigma)_{nat}$ is also a term in $(T_{\Sigma B})_{nat}$; therefore every term in $(T_\Sigma)_{ecol}$ is also a term in $(T_{\Sigma b})_{ecol}$ and $\langle x,top(t)\rangle +1_e t'$ is a E-normal form.

Proposition 5:

Let **SRelation1** be the parameterised specification, **StackB** be the base specification and **Stack** the parameter specification. If $E \backslash E_B$ is terminating, then **SRelation1** is persistent w.r.t **Stack + Nodes&Edges(Nat)**.

proof: by Persistency Theorem II

i) There are no $stack, nat$ or $bool$ sorted operations in $\Sigma_B \backslash \Sigma_s$.

ii) There are no $stack, nat$ or $bool$ sorted equations in $E_B \backslash E_s$.

iii) All the l.h.s.'s of equations in $E \backslash E_B$ contain at least one symbol from $\{\Downarrow, \Rightarrow, abs, rep\}$ and no symbols from $\{0, succ, push, create, top, pop\}$.

iv) $E \backslash E_b$ is linear and E-normalising because by Propositions 3 and 4, for all $t \in (T_\Sigma)_{stack}$, $rep(t) \Rightarrow_E [\Downarrow t, \Rightarrow t]$, $abs(rep(t)) \Rightarrow_E t$ and $\Downarrow t$ and $\Rightarrow t$ have

E-normal forms.

v) Using the definitions and the fact that the outermost symbol of every l.h.s of $(E \setminus E_B)$ is a symbol from $(\Sigma \setminus \Sigma_b)$, we can show, in a manner similar to Example 5.13 in [Pad 85], that every critical pair of $(E \setminus E_B)$ into $(E \setminus E_B)$ and every recursive critical pair of $(E \setminus E_B)$ into E_B is $(E \setminus E_B)$-convergent, and there are no parallel critical pairs of E_B into $(E \setminus E_B)$ □

4.4 Summary

In this chapter we have considered the syntactic presentation of the storage relations of an object (keyless/implicitly keyed) specification.

First, we considered a formal approach in which the specification of storage relations is parameterised by a (representation of) the object specification. This approach was not successful because we cannot axiomatise the relations within the standard equational framework. This result does not affect our methodology, but it does restrict the potential for automating the implementation process. If, in the future, we can define suitable extensions to the *process* specification language (as opposed to the *object* specification language), then we may be able to automate this part of the methodology.

Second, we presented a less formal, hierarchical approach in which the object specification is enriched with new sorts, operations and equations. The resulting specification is called **`SRelation`**. Only part of the enrichment is formally specified, although we demand that the resulting **`SRelation`** fulfill certain requirements. This means that for a given object specification, the specification of its storage relations is not necessarily syntactically or semantically unique. The chapter concludes with some examples of **`SRelation`** specifications which illustrate the approach and a discussion on proving the correctness of the **`SRelation`** specifications.

Chapter Five
Storage Graphs

In this chapter we begin to construct the implementation of an keyless/implicitly keyed ADT in a stepwise manner.

The storage relations of an ADT are extended to the *storage graphs* of an ADT. A storage graph is a directed graph with some additional information about which nodes, the *access nodes*, should be efficiently accessible at any time. A storage graph depends on a term of the derived sort: the labels of the nodes are given by the contents set and the edges are given by the storage relation. The (labels of the) access nodes are given by primitive sorted operations called *access operations*; these operations may include derived operations.

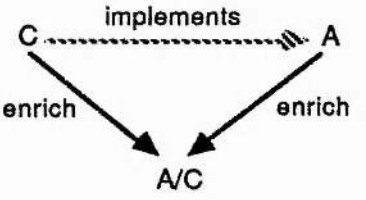
We discuss the criteria for selecting the access nodes and we give two strategies for defining the access operations: the first is a general, automatic strategy; the second may require human intervention. The latter strategy involves considering storage relations as representations of elements of an ADT; given an object ADT and the specification of its storage relations, we construct a partial implementation of the former by the latter. and define the access operators according to some aspects of the (partial) implementation.

When the access operations are specified for the given object ADT (by either strategy), then we specify the class of storage graphs as an ADT and construct an implementation of the former by the latter.

5.1 A Stepwise Approach to Constructing Implementations

Implementing an ADT: choosing the data structure to represent elements of the derived sort and constructing the functions and procedures to implement the operations, is very complex when efficiency is important. Our method constructs the implementation in a *stepwise* manner.

When a programming language has a well-defined denotational or algebraic semantics, then the language can be specified as an ADT. The usual notion of implementation for an ADT is expressed as a relationship *between* ADTs [Hoa 72], [ADJ 78], [EHK 82], [Ehr 82], [Bro 84]. Informally, a specification C (the "concrete" ADT) implements a specification A (the "abstract" ADT) when the (visible) operations of A can be *associated* with (derived) operations in C which *realise* the behaviour defined by the equations in A. In our context, implementation is essentially the process of imposing the structure of (the initial algebra) of A onto the structure of (the initial algebra) of C. When we enrich C with operator symbols for the derived operators and the corresponding equations, then we have another specification which we call A/C (read as "A over C"). We refer to C as the *implementing* ADT, A as the *implemented* ADT, and A/C as the *implementation* ADT.



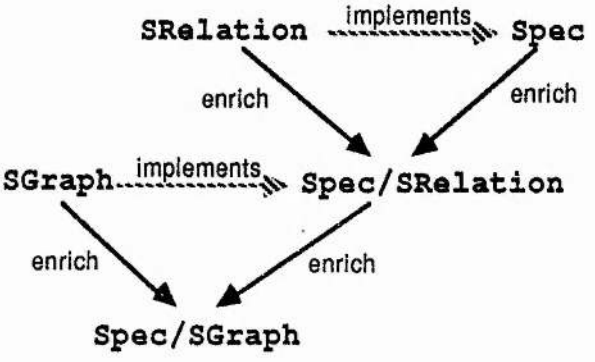
Of course the relationships between A, C, and A/C, (morphisms), must satisfy certain requirements in order for A/C to be a correct implementation; these requirements will be discussed further in section 4.3.

At the beginning, we have the problem of implementing a completely unfamiliar, arbitrary ADT. Given such an object ADT, call it **Spec**, we can specify the class of storage relations of **Spec** as the ADT **SRelation**. When we consider the storage relation at t as a *representation* of the **Spec** element [t], and enrich **SRelation** with implementations of the operations of **Spec** (i.e. when we construct **Spec/SRelation**), then we reduce the problem to that of implementing binary relations, or directed graphs.

The classification of an ADT by storage type is useful when we consider the problem of implementing **Spec/SRelation**, but, assuming that mutable data structures are going to be used, this information alone is not adequate for the construction of an *efficient* implementation.

Recall that the crucial decision when choosing either array-based or linked data structures is the choice of entry points into the array or linked structure. The number and nature of these entry points can greatly affect the efficiency of the implementation. In order to make these decisions (at some later stage), we define, for each ADT, a set of selectors (consisting of given and derived operators) which return the set of nodes (labelled by primitive sorted elements in $\Downarrow t$) which should be regarded as entry points into the digraphs which represent the storage relations. These designated selectors are called *access operators*; for each element of the derived sort, the designated nodes are called *access nodes*. The access nodes are chosen according to two criteria: 1) all the nodes in the digraph represention of a storage relation which need to accessed (when implementing the operations of **Spec** by **SRelation**) should be reachable from the access nodes, and 2) the path (from the access nodes) to those nodes we need to access should be efficient.

The *storage graph* at t is a tuple which consists of the contents set of t, the storage relation at t, and the (labels of the) access nodes of t. Given an object ADT **Spec**, we can specify the class of storage graphs of the elements of **Spec**; we call this latter ADT **SGraph**. When we consider the storage graph at t as a *representation* of the storage relation at t, and enrich **SGraph** with implementations of the operations of **Spec/SRelation** (i.e. we construct **Spec/SGraph**), then we reduce the implementation problem to that of implementing directed graphs with nodes designated for efficient access. For each ADT **Spec**, we will base our implementation decisions on its storage type and the properties of **Spec/SGraph**. In chapter six we discuss how to choose data structures and implement **Spec/SGraph** using those data structures; the aim of this chapter is to discuss the construction of **Spec/SRelation** and **Spec/SGraph** given an ADT **Spec**. The stepwise implementation is summarised by the following diagram:



5.2 Implementation

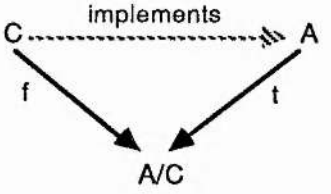
In this section we give some standard definitions concerning the implementation of ADTs. There are many notions of implementation in the literature: for example, [Ehr 82] and [EKM 82] give definitions assuming parameterised specifications with initial semantics, [SaW 82] gives definitions assuming parameterised specifications with loose semantics, and [Bro 84] gives definitions assuming hierarchical specifications with initial semantics. Our main concern is with the syntactic aspect of implementation (i.e. the relationships between specifications), and that along with the notion of correctness, we have some method or guidance for constructing correct implementations; i.e. given ADTs A and C, we can construct A/C such that it is the correct implementation of A. Moreover, the implementations we construct should be composable.

Although some of the categorical approach of Ehrich [Ehr 82] is too elaborate for our purposes (because we only consider hierarchical specifications), we adopt this notion of correctness for the following reasons: it allows us to construct implementations using the inductive theory of the implementing ADT, it allows "redundant" elements in the implementation ADT (elements which are not representations of the implemented ADT), it allows the introduction of arbitrary recursion schemes for specifying derived operators, and it guarantees the composability of implementations.

Before giving the formal definition of a *full implementation* from [Ehr 82], we explain the motivation for the definition.

Consider specifications A and C. Informally, C implements A when we can represent the sorts and operations specified in A by the sorts and (derived) operations specified in C. The specification consisting of C enriched with derived operations (i.e. some operator symbols and equations) is called A/C. The implementation relationship can be described as a relationship between specifications: we must have a signature morphism mapping the sorts and operator symbols of A to the sorts and operator symbols in A/C, (these are typically derived operators of C), and every valid equation in the equational theory of A must be provable (after renaming) in the inductive theory of C. Implementation is essentially a pair of morphisms with a common target; namely, A/C is related to A and C by two morphisms. One morphism, f (for "full"), maps C to A/C ; the other morphism, t (for "true") maps A to A/C. In this context, a morphism is mapping between initial algebras and is given by a pair which consists of a map

between the sorts and a family of sorted maps between the carriers. Full definitions of the elementary category theory used in this section and the category of specifications **spec** are given in Appendix One; here, we consider the properties of f and t which make A/C a correct implementation.

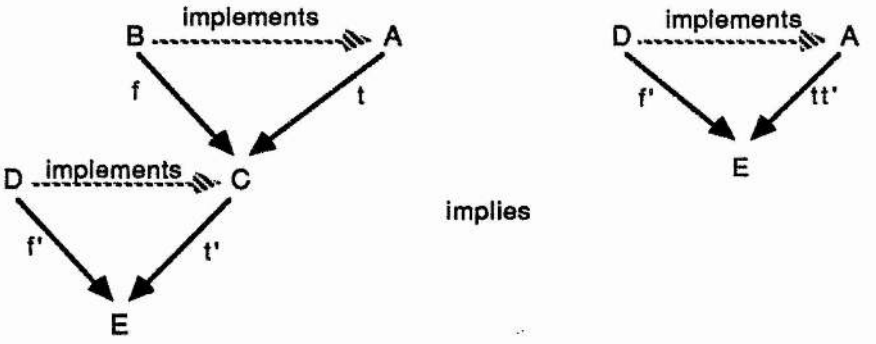


First, the morphisms should describe, up to renaming, sub-specification relations; that is, up to renaming, the sorts, operators and the equational theory of A and C should be contained in the sorts, operators, and equational theory of A/C. Second, the morphism f should not introduce any new sorts, nor any new elements into A/C which are not already specified by C. This latter requirement is usually referred to as *sufficient-completeness* (cf. chapter 2). f may however introduce new equalities between the elements specified by C. Third, the morphism t may rename the sorts and operators of A, but it should be not introduce into A/C any new equalities between (the representations of) the elements specified by A. This requirement is usually referred to as *consistency*. This is similar to hierarchical consistency (cf. chapter 2) but the requirement is now extended to all sorts.

The requirement for the sub-specification relation is ensured by forcing f and t to be embeddings; a morphism (h,g) between Σ-algebras is an embedding when h and g are both injections. The requirements for f (i.e. no new sorts and sufficient-completeness) are ensured when f is a full Σ-embedding; the requirement for t (i.e. consistency) is ensured when t is a true embedding.

Definition: From [Ehr 82]
A ***full implementation*** of specification A by specification C is a a triple (A/C,f,t) where f: C -> A/C is a full Σ-embedding (where C=(Σ,E)) and t: A -> A/C is a true embedding. We say that C ***implements*** A when there is a full implementation of A by C.

Full implementations are composable, i.e.



Proposition: From [Ehr 82]
Let A,B,C, and D be specifications, f,f',t, and t' be morphisms, and (C,f,t): B -> A be a full implementation. If (E,f',t'): D -> C is a full implementation, then (E,f',tt'): D -> A is a full implementation.

5.2.1 Proving an Implementation Correct

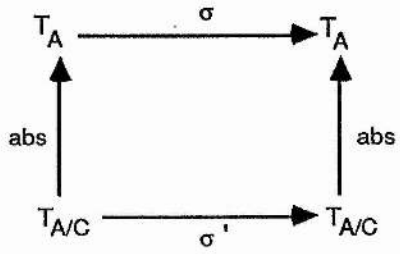
Given an implemented ADT A and an implementing ADT C, if we construct A/C by enriching C with derived operators and equations such that the resulting specification is sufficiently-complete (w.r.t. C), then clearly there is a full Σ-embedding from C to A/C (where Σ is the signature of C) and it only remains to show that there is a true embedding from A to A/C. This means that we have to show that the equations in the equational theory of A are valid, up to renaming, in the equational theory of A/C and vice-versa. For example, given a signature morphism (h,g) from the signature of A to the signature of A/C which maps $\sigma: s_1 \rightarrow s_1$ to $\sigma': h(s_1) \rightarrow h(s_2)$, then $\sigma'(x)$ and $\sigma'(y)$ must be in different congruence classes only when $\sigma(x)$ and $\sigma(y)$ are in different classes. The traditional way to show this property is to give a *representation* mapping [ADJ 78], [EKM 82], or a pair of *representation* and *abstraction* mappings rep and abs [Dar 82], [Bro 84], between the models of A and A/C (T_A and $T_{A/C}$ resp.) with the following relationships:

$rep: T_A \rightarrow \mathcal{P}(T_{A/C})$

$abs: T_{A/C} \rightarrow T_A$

(I1) $(\forall x \in T_A)\ abs^{->}(rep(x)) = \{x\}$

(12) rep is total and abs is partial, surjective and homomorphic; i.e. the following diagram commutes:



We always assume that rep is the identity mapping, (i.e. rep(x)={x}) on the elements specified by the primitive part of T_A. The "relational" nature of rep can cause problems because it means that [abs(x)]=[abs(y)] does not imply [x]=[y]. Fortunately, we can often allow abs to induce a congruence on the elements of T_C in $T_{A/C}$ which does not violate the sufficient-completeness condition. We can do so because such a congruence is often already valid in the inductive theory of C.

In addition, we have to consider the partial ordering on sorts: the signature morphism must be monotonic, i.e. if $s_1 \leq s_2$ then $h(s_1) \leq h(s_2)$. The morphism is not required to be an injection and so partially-ordered sorts may be collapsed into one sort; i.e. we may have $h(s_1) = h(s_2)$ when $(s_1 \leq s_2)$ but $(s_1 \neq s_2)$.

When we have a specification of the abstraction mapping, the equations specifying the implementations of the operators in the implemented ADT, (i.e. the derived operators) can be synthesised as theorems of the implementing ADT. In [BuD 77] and [Dar 82] it is possible, sometimes, to construct an implementation from the implementing ADT using a specification of abs and "program transformation" techniques. We say "sometimes" because their rules only synthesise implementations in the equational theory of the implementing ADT and such implementations do not always exist. When the specifications of both the implemented and the implementing ADT can be organised into confluent, terminating rewriting systems, then an implementation in the equational theory of the implementing ADT can be synthesised automatically using the equational strategy given by Kapur and Srivas in [KaS 85]. Unfortunately, there is no such general inductive strategy.

In the following section we give seven "program transformation" rules for enriching the implementing specification.

5.3 Constructing a Correct Implementation

Given an implementing ADT $C=(\Sigma_C,E_C)$, an implemented ADT $A=(\Sigma_A,E_A)$ and a representation mapping rep: A -> C, we can construct an implementation A/C by applying the inference rules given below. The inference rules describe the circumstances under which the implementing ADT can be enriched with new operators and equations. Of course we must first enrich the signature of C with the abstraction operator abs and the representations of the operators in A i.e. for every operator $f \in \Sigma_A$ with arity $s_1,..,s_n->s$, we add the operator rep(f): rep (s_1),.., rep (s_n) -> rep(s).

The following rules are similar to those given in [Dar 82] and [KaS 85] although we do not assume that all elements in the implementing ADT are representations; i.e. rep is not surjective and so there is a non-trivial representation invariant. [Dar 82] does not allow the introduction of equations from the inductive theory and [KaS 85] does not discuss the explicit introduction of hidden operations. [KaS 85] does allow the introduction of equations from the inductive theory because the equations are regarded as rewrite rules; the validity of the new equations can be effectively checked with the Knuth-Bendix completion procedure (i.e. by proofs by consistency).

We assume that the variables occurring in an equation can be (safely) renamed at any time and that in each equation, variables with different sorts have different names. Rules 1-6 describe the enrichment of (Σ_C,E_C) by equations only; rule 7 describes the enrichment of (Σ_C,E_C) by operators and equations.

Rules

1) **abstraction specification**:- introduce a new equation, the specification of rep(f).

$$f \in \Sigma_A \text{ with arity } s_1,..,s_n->s,\quad rep(f)=F,\quad x_1:s_1,..,\ x_n:s_n \text{ are variables}$$

$$abs(F(rep(x_1),...,rep(x_n))) = f(abs(rep(x_1)),....,abs(rep(x_n))) \in E_C$$

2) **instantiation**:- introduce a substitution instance of an existing equation.

$$\frac{\alpha = \beta \in E_C,\ x \text{ is a variable},\ t \text{ is a term}}{\alpha[t/x] = \beta[t/x] \in E_C}$$

3) **unfolding**:- replace an instance of the l.h.s. of an existing equation by the corresponding instance of the r.h.s.

$$\frac{\alpha_1 = \beta_1,\ \alpha_2 = \beta_2 \in E_C,\ \alpha_2 \text{ occurs in } \beta_1}{\alpha_1 = \beta_1[\beta_2/\alpha_2] \in E_C}$$

4) **folding**:- recursion and function introduction, replace an instance of the r.h.s. of an equation by the corresponding instance of the l.h.s.

$$\frac{\alpha_1 = \beta_1,\ \alpha_2 = \beta_2 \in E_C,\ \beta_2 \text{ occurs in } \beta_1}{\alpha_1 = \beta_1[\alpha_2/\beta_2] \in E_C}$$

5) **abs-dropping**:- drop abs from the l.h.s. and the r.h.s. of an existing equation.

$$\frac{\text{abs}(\alpha) = \text{abs}(\beta) \in E_C,\ \alpha = \beta \in \text{inductive theory of } E_C}{\alpha = \beta \in E_C}$$

6) **introduction**:- introduce an arbitrary equation from the inductive theory.

$$\frac{\alpha = \beta \in \text{inductive theory of } E_C}{\alpha = \beta \in E_C}$$

7) **hidden specification**:- introduce a hidden operation f.

$$f: s_1,...,s_n \rightarrow s,\ e_1,...,e_m \text{ are equations,}$$

$$(\Sigma_C \cup \{f\}, E_C \cup \{e_1,...,e_m\}) \text{ is sufficiently-complete and consistent w.r.t. } (\Sigma_C, E_C)$$

$$f \in \Sigma_C,\ \ e_1,...,e_m \in E_C$$

Rules 1-4 introduce equations valid in the equational theory of the implementing ADT, whereas rules 5-6 introduce equations valid in the inductive theory of the implementing ADT. Rules 1-5 introduce equations which are derived from existing equations. We note that rule 5 is a restricted case of rule 6 and is most likely to be applicable when α does not contain any operators from the implemented ADT. We have not explicitly discussed how equations are shown to be valid in the inductive theory but clearly an induction rule based on the structure of ground terms is required as the usual rule:

$$\frac{\text{for every ground substitution } \sigma,\ \sigma(t)=\sigma(u) \in \text{equational theory}}{t=u \in \text{inductive theory}}$$

has infinitely many premises and is not therefore effective.

Given specifications A, C, a signature morphism σ, and the abstraction mapping abs, we say that the specification C enriched according to these rules is *well-spanned with respect to the representation* when for each operator $f: s_1 .. s_n \rightarrow s$ in Σ_a, for all ground generator terms $t_i:(s_i)$, $(1 \leq i \leq n)$, $rep(f)(rep(t_1),..,rep(t_n))$ is an instance of the l.h.s. of an equation. Clearly, a desirable implementation of A is an enrichment of C (constructed using only these rules) which is well-spanned with respect to the representation and the equations do not contain operator symbols from A. We note that when the representation mapping is not surjective, then the resulting specification is not well-spanned and thus the new operations may be only partially specified.

5.4 Access Nodes and Operators

We now consider the representation of a storage relation by a digraph. In order to choose (eventually) a representation of digraphs in a programming language, we designate a subset of the nodes of a digraph as ***access nodes***. These are nodes which may be regarded as entry points into the digraphs; for each digraph which represents a storage relation, the labels of the access nodes are elements of the corresponding contents set.

The function of the access nodes is twofold; when we consider a storage relation as a digraph, then the set of access nodes should: 1) ensure that all the nodes in the digraph which need to be accessed are accessible (from the set of access nodes), and 2) define a potential space-time tradeoff. Namely, a node may be designated an access node if the inclusion of the node would enable operations on the digraph to be more (time) efficient; such nodes may already be accessible via the other access nodes. (Here, our notion of time efficiency is related to the number of edges traversed.)

Of course if the digraph is small enough, then the entire node set can be considered as access nodes. However, in most ADTs, the digraphs which represent the storage relations eventually become so large that this is no longer practicable; we must select the access nodes according to the above criteria.

So, we must be able to select the access nodes of a given digraph representation of a storage relation; moreover, we would like a general description, or specification, of the access nodes which does not depend on a particular digraph. (Although it is quite reasonable to expect that the selection depends on some of the properties which hold for all of the digraph representations of the storage relations of the object ADT.)

From our observational point of view, we are interested in the *labels* of the access nodes; i.e. subsets of the domains of storage relations. The (labels of the) access nodes can be specified either by operations on digraphs (for example, an operation which returns the maximal node), or by operations on the derived sort elements from which the storage relation and digraphs are derived (for example, the operation `front`). We assume the latter form of specification: selectors in the object specification; we call the operators which name these operations ***access operators***. Thus, we may refer to the access operators of a specification.

In the following sections, we consider the specification of the access operators according to the criteria described above. Because there are infinitely many operations which are applicable to a digraph, we cannot, in general, specify the access operators which will define a reasonable space-time trade-off. Instead, we give two different strategies for determining the access operators.

The first is a general, "automatic" strategy which does not depend on the object ADT; however, because of this, it does not incorporate any aspects of efficiency analysis. The strategy uses a general result about the reachability of elements in the domain of the transitive closure of a storage relation. The second strategy depends on the object ADT; some aspects of efficiency analysis are incorporated by considering the implementation of the object operations on the digraph representations of the storage relations. Because this strategy involves implementation, and hence some aspects of the inductive theories, some human intervention may be required.

5.5 Defining Access Operators: Method I

This method defines the access nodes of a digraph representation of a storage relation to be the least subset of nodes which ensures that the first criterion is fulfilled: all the nodes in the digraph are reachable from the access nodes. The method is a general one; that is, it does not depend on the particular object specification. Hence, for a given specification, the method may not return the least set which satisfies the first criterion. We show that for each storage relation $\Rightarrow_t$, all the elements in the domain of $\Downarrow t$ are reachable from the set of primitive elements returned by the application of the selectors to t, and the application of the compostition of the selectors with the rearrangers to t. The set of nodes thus defined is similar to the graph-theoretic notion of a root, or a point basis [Har 69], except that it is not required to be minimal.

Using this method, we define set of access operators of an ADT to be the set of selectors and compositions of selectors with rearrangers. In the following definition, we assume that $f \circ g(x) =_{def} f(g(x))$.

Definition: Let (Σ,E) be a keyless/implicitly keyed specification with selectors $s_1,...,s_n$ and rearrangers $r_1,...,r_m$. The ***access operators*** of (Σ,E) are the selectors s_i and the primitive sorted operators s_{ij}, defined as $s_{ij} =_{def} s_i \circ r_j$, for $1 \leq i \leq n$, $1 \leq j \leq m$.

For a given storage relation $\Rightarrow_t$, the (labels of the) ***access nodes*** of $\Rightarrow_t$ are the set of primitive elements $\{[s(t)] \mid s \in (\{s_i\} \cup \{s_{ij}\})\}$.

Lemma: Let (Σ,E) be a keyless/implicitly keyed specification, let t be a term of the derived sort, let $(\Rightarrow_t)^+$ be the transitive closure of $\Rightarrow_t$, and let A be the set of labels of the access nodes of $\Rightarrow_t$. For every element x in $\Downarrow t$ which is not in A, there is a path in the storage relation at t from an element in A to x. i.e.

$$(\forall x \in \{z|\})\ (\exists y \in A)\ y\ (\Rightarrow_t)^+ x\ .$$

Proof: Let x be an element in $\Downarrow t$ which is not the (label of) an access node. By the defn. of $\Downarrow t$, when $x \in \Downarrow t$ and $x \notin A$, then there is a selector σ_s and a class C in $\downarrow t$ such that $\sigma_s(C)=x$. By the defn. of $\downarrow t$ and because we do not consider elements of A in $\Downarrow t$, there is a t' in $\downarrow t$ and n eliminators $\sigma_1,...,\sigma_{n'}$ where $n \geq 1$, such that $\sigma_n(...\sigma_1(t')...)$ is in C . Either [t] = [t'], or there is a rearranger σ_r such that $[t']=[\sigma_r(t)]$. By the defn. of $\rightarrow_t$, $[\sigma_{n-1}(...\sigma_1(t')...)] \rightarrow_t C$, $[\sigma_{n-2}(...\sigma_1(t')...)] \rightarrow_t [\sigma_{n-1}(...\sigma_1(t')...)]$, and so on until $[t'] \rightarrow_t [\sigma_1(t')]$. By transitivity, we can show that $[t'](\rightarrow_t)^+ C$. Also by transitivity, we can show that $[\sigma_s(t')]\ (\Rightarrow_t)^+\ \sigma_s(C)$. $\sigma_s(C)=x$, and so by substitution, $[\sigma_s(t')]\ (\Rightarrow_t)^+\ x$. By the definition of A, $A=\{[\sigma_s(t)],[\sigma_s(\sigma_r(t))]\}$; if $[t']=[t]$ then y is $[\sigma_s(t)]$ otherwise y is $[\sigma_s(\sigma_r(t'))]$. □

In some specifications, the access nodes will always be a point basis. For example, they will always be a point basis in **Queue** and **Stack**, but not in **Reversible_List_1** nor in **Circular_List_right** because in these specifications, the set of access nodes is not minimal. We conclude this section with some examples of access operators defined by this method.

Example 5.1: Stack
The access operators are {top}.

Example 5.2: Queue
The access operators are {front}.

Example 5.3: Binary_Tree
The access operators are {root}.

Example 5.4: Reversible_List_1
The access operators are {hd, hdrev}.

Example 5.5: Circular_List_right
The access operators are {hd, hdshift}.

5.6 Defining Access Operators: Method II

Although we have said that in general, there are infinitely many operations which are applicable to digraphs, we are only interested the operations which are *implementations* of operations in the given object specification. By considering the implementation of the object specification by its storage relations, we can determine exactly which primitive sorted elements need to be accessed efficiently. Namely, these are the (classes containing) terms which occur (as subterms) in the equations which define the implementations of the operations of the object ADT. These elements denote *positions* in the storage relations and so efficient access to these elements would improve the (time) efficiency of the storage relation operations; that is, it would improve the efficiency of the *graph* operations which are representations of operations from the object specification. Often, these elements are defined by additional (auxiliary, hidden) operations which are synthesised in order to implement the object specification by its storage relations. For a given storage relation, we insist that all these elements are given

by primitive sorted operations (as opposed to compositions of selectors and other operations); this is ensured by constraining the form of the implementation. The constraint is described in the following sections.

In this method, the access operators of an ADT are defined by the given selectors of the object specification, together with any additional selectors synthesised during the (partial) implementation of the object specification by its storage relations. The motivation for considering partial implementations, and the nature of the partiality, is described in the following section.

5.6.1 Implementing Object Specifications by Storage Relations

This method of determining the access nodes and operators depends on the analysis of the operations on storage relations and contents sets which can be regarded as implementations of operations on elements of the derived sort. Again, in the analysis, we encounter a situation which is similar to the one described in §3.2: the equivalences between primitive sorted subterms of terms of sort *ncol* and *ecol* interfere with our analysis. In this situation, such equivalences do not affect the results of the analysis, but they do complicate the analysis. The ADT **Col** is an explicitly keyed ADT. If, during the implementation, the specification of **Nodes&Edges** needs to be enriched with operations which select or remove items from certain positions in the contents set or storage relation, then such operations are much easier to specify when each δ-sorted subterm can be uniquely described (up to equivalence in the inductive theory) as the application of a (possibly auxiliary) selector to the derived term which the *ncol* or *ecol* term represents. Namely, we would like to have an injection between the δ-sorted subterms of ground generator terms of the derived sort and the δ-sorted subterms of the ground generators of sort *ncol* or *ncol*. Complications are caused by the fact that in the standard model, we cannot "remember" how we retrieved a data item; i.e. we can have the situation that the term `s(e(e(t)))` is identified with the term `s(e(t))`, where *s* is a selector and *e* an eliminator. Some examples in §5.6.3 will illustrate the complications caused by this situation. Perhaps, again, we should consider a trace algebra as suggested in §3.2. A less elaborate alternative is to consider, as we did in §3.2, only terms with distinct primitive sorted sub-terms.

Somewhat informally, for every **SRelation** derived from **Spec**, we enrich **SRelation** with the specification of **leaves**:τ -> ncol, (as given in §2.5), and the following operators and equations:

```
multiple  : ncol -> bool
mult      : δ ncol -> bool
ok        : ncol -> bool

                multiple([]_n)          = F
∀n:δ,N:ncol.    multiple(n +1_n N)      = mult(n,N) ∨ multiple(N)
∀n:δ.           mult(n,[]_n)            = F
∀n,n1:δ,N:ncol. mult(n,n1 +1_n N)       = (n=n1)∨ mult(n,N)
∀N:ncol.        ok(N)                   = ~(multiple(N))
```

As a formal specification, this enrichment depends on an object specification for an instantiation of τ and δ. Informally, we refer to the enrichment as **Filter**, and when given in the context **SRelation + Filter**, where **SRelation** is the specification of the storage relations derived from **Spec**, then we assume that τ and δ are instantiated by the derived and primitive sorts resp. of **Spec**.

In conclusion, for the purposes of determining the access operators, we consider only the implementations of derived sort terms t s.t. ok(t) $=_E$ T. We refer to these implementations as the *partial* implementations.

5.6.2 Constraining the Form of Implementations

Consider the (partial) implementation of an object specification **Spec** by the specification **SRelation + Filter** derived from **Spec**. Trivial implementations can always be constructed automatically because the representation and abstraction mappings, the operators *rep* and *abs* given in §4.2.3, are predefined independently of the object specification.

Trivial implementations are possible because **Spec** is actually embedded in **SRelation**. For example, when **Spec**=(Σ,E) and **SRelation**=(Σ',E'), then the representation of an operator f: s_1 ... s_n -> $\tau \in \Sigma$, F: rep(s_1) ... rep(s_n) -> rep(τ) $\in \Sigma'$, is specified by:

$$
\begin{aligned}
&\mathtt{abs}(\mathtt{F}(\mathtt{rep}(t_1),\ldots,\mathtt{rep}(t_n))) \\
&\qquad = \mathtt{f}(\mathtt{abs}(\mathtt{rep}(t_1)),\ldots,\mathtt{abs}(\mathtt{rep}(t_n))) && \text{abs spec.}\\
&\qquad = \mathtt{f}(t_1,\ldots,t_n) && \text{unfold}\\
&\qquad = \mathtt{abs}(\mathtt{rep}(\ \mathtt{f}(t_1,\ldots,t_n\ ))) && \text{fold}\\
&\qquad = \mathtt{abs}[\Downarrow \mathtt{f}(t_1,\ldots,t_n), \Rightarrow \mathtt{f}(t_1,\ldots,t_n)] && \text{unfold}
\end{aligned}
$$

And so when we drop abs, we have

$$(*) \qquad \mathtt{F}(\mathtt{rep}(t_1),\ldots,\mathtt{rep}(t_n)) = [\Downarrow \mathtt{f}(t_1,\ldots,t_n), \Rightarrow \mathtt{f}(t_1,\ldots,t_n)]$$

In essence, the third component of **SRelation** is just an application of the hidden specification rule (cf. §5.3).

The above specification of F is not very useful because it does not implement f using the operations of **SRelation**; it uses only the **Spec** part of **SRelation**. Instead, we want to impose the structure of **Spec** onto the **Nodes&Edges** component of **SRelation**. Namely, we want to find specifications for $F(x_1,\ldots,x_n)$, where $x_1,\ldots,x_n$ are in the domain of the relevant abstraction mappings; i.e. we want equations of the form (*) in which the τ-sorted operators of **Spec** do not appear. The δ-sorted operators of **Spec** may appear because *rep* is the identity mapping on these sorts; specifications for the (representations of the) primitive sorted operators therefore retain the form of (*).

When f is a τ or ϕ-sorted operator, then we must synthesise a specification of F in which the derived sort operators of **Spec** do not appear; we look for a definition of g such that

$$(**) \quad \mathtt{F}(\mathtt{rep}(t_1),\ldots,\mathtt{rep}(t_n)) = \mathtt{g}(\mathtt{rep}(t_1),\ldots,\mathtt{rep}(t_n))$$

where $t_1,\ldots,t_n$ are τ or ϕ-sorted terms and g is defined using only $t_1,\ldots,t_n$ and the language of **SRelation** without the τ and ϕ-sorted operators. If we may regard **SRelation** as a *module*, then the module does not export the sort τ and its subsort ϕ. We refer to the exported language of this module as the *restricted* language of **SRelation**.

The theory of **Nodes&Edges** given in §4.2 is not very expressive; in fact, it is anarchic. An implementation in the restricted language of **SRelation** will require an enrichment (i.e. hidden specification) of the theory of *ecol*, *ncol* and *graph*. We

constrain the enrichments by disallowing any operations with arity `ncol -> ncol`, `ecol -> ecol`, `graph -> graph` or `graph -> δ`, which are not (in the last case) representations of operations from the object specification. Moreover, all auxiliary δ-sorted operators must have arity s -> δ, where s ∈ {ϕ,τ,graph}. This constraint ensures that `ncol` and `ecol` remain explicitly keyed ADTs; i.e. the access to the nodes and edges therein must be specified explicitly.

In the following section we give a "standard" enrichment of **`SRelation`** which is sufficiently-complete and consistent w.r.t. **`Nodes&Edges`**. The choice of operations for the enrichment is of course arbitrary; we have chosen operations which are either "standard" selectors (i.e. inverses to constructors) or typical digraph operations which can be implemented in constant time in a pointer implementation of digraphs. These latter operations will be useful in the following chapter when we synthesise pointer implementations.

5.6.2.1 The Enriched Nodes&Edges

```
Col'(D:U) =

data sorts col

gen
[]      : col
+l      : elem col -> fcol

opns
_\_     : col elem -> col                    !remove elem
_\\_    : col col -> col                     !remove elem
+r      : elem col-> fcol                    !insert right

eqns
∀C:col.              [] \ C          = []                          C1
∀C:col,x,y:elem.     (x +l C) \ y    = C                 if(x=y)   C2
∀C:col,x,y:elem.     (x +l C) \ y    = x +l (C \ y)      if ~(x=y) C3

∀x:elem.             x +r []         = x +l []                     C4
∀C:col,x,y:elem.     x +r (y +l C)   = y +l (x +r C)               C5

∀C:col.              C \\ []         = []                          C6
∀C,C1:col,y:elem.    C \\ (y +l C1)  = (C \ y) \\  C1              C7
end
```

```
Nodes&Edges'(D:U) =
enrich
    derive sorts ncol,

    gen   []n :ncol, _+ln_: elem ncol -> ncol
    opns  _+rn_ : elem ncol -> ncol, _\n_ : ncol elem -> ncol,
          _\\n_ : ncol ncol -> ncol,
    from Col'(D) by
          ncol is col, []n is [], +ln is +l,
           +rn is +r, \n is \, \\n is \\
    +
derive sorts ecol

    gen   []e :ecol, _+le_: elem ecol -> ecol
    opns  _+re_ : elem ecol -> ecol, _\e_ : ecol elem -> ecol,
          _\\e_ : ecol ncol -> ecol

    from Col'((Pair(D))[elem is pair]) by
          ecol is col, ecol is col,[]e is [], +le is +l,
          +re is +r,\e is \, \\e is \\
by

data sorts graph, rgraph
subsorts rgraph ≤ graph

opns
s       : pair -> elem                  !source node of edge
t       : pair -> elem                  !target node of edge
adj     : ecol elem -> ncol             !nodes adjacent to node
addto   : ecol ncol elem -> ecol        !add edges to node
addfr   : ecol ncol elem -> ecol        !add edges from node
scen    : ecol elem -> ecol             !edges with node as source
tgtn    : ecol elem -> ecol             !edges with node as target
mapt    : ecol elem -> ncol             !nodes with node as target
[_,_]   : ncol ecol ->graph
```

eqns

```
∀E:ecol,x:pair,y:elem.
scen(x +l_e E,y)          = x +l_e scen(E,y)        if (s(x)=y)     A1
∀E:ecol,x:pair,y:elem.
scen(x +l_e E,y)          = scen(E,y)               if ~(s(x)=y)    A2
∀y:elem. scen([]_e,y)     = []                                      A3

∀E:ecol,x:pair,y:elem.
tgtn(x +l_e E,y)          = x +l_e tgtn(E,y)        if (t(x)=y)     A4
∀E:ecol,x:pair,y:elem.
tgtn(x +l_e E,y)          = tgtn(E,y)               if ~(t(x)=y)    A5
∀y:elem. tgtn([],y)       = []_e                                    A6

∀E:ecol,x:pair.
mapt(x +l_e E)            = map(tgtn(x +l_e E))                     A7
∀E:ecol,x:pair.
map(x +l_e E)             = t(x) +l_n map(E)                        A8
mapt([]_e)                = []_n                                    A9
map([]_e)                 = []_n                                    A10

∀E,T:ecol,ncol,x,y:elem.
addto(E,x +l_n T,y)       = addto(<x,y> +l_e E,T,y)                 A11
∀E:ecol,y:elem.
addto(E,[]_n,y)           = E                                       A12
∀E,T:ecol,ncol,x,y:elem.
addto(E,x +l_n T,y)       = addfr(<y,x> +l_e E,T,y)                 A13

∀E:ecol,y:elem.
addfr(E,[]_n,y)           = E                                       A14
∀E:ecol,x:elem.
adj(E,x)                  = mapt(scen(E,x))                         A15
```

end

Again, a specification language with higher order functions would simplify this and subsequent specifications. In later sections, we refer to equations "Nn" and "En" as the instantiations of equation "Cn" , $1 \leq n \leq 8$, with *ncol* as *col* and *ecol* as *col* resp.

5.6.3 Synthesising Implementations for Derived Sort Operators

The specification of an implementation of a derived sort operator in the form of (**) can be synthesised from a specification in the form of (*) by applying the rules given in §5.3. Each synthesis can be characterised by the rules which are applied; we

distinguish three distinct kinds of synthesis strategies. These are described below and we discuss how, with increasing difficulty, the synthesis strategies might be automated.

When only rules 1-5 are used, then the implementation uses just one particular kind of property from the inductive theory of the implementing ADT; when the specifications can be organised into confluent, terminating term rewriting systems, then the implementation may be synthesised automatically using the method of [KaS 85].

When rules 1-6 are used, then the implementation may use any of the properties from the inductive theory of the implementing ADT and so the implementation cannot be synthesised automatically by the method of [KaS 85]. However, if the specifications fulfill certain requirements, then the results of [Lan 87] show that an implementation may be synthesised automatically using an inductive inference algorithm.

When all of the rules are used, then the implementation may not only use any of the properties from the inductive theory of the implementing ADT, but it may also increase the expressiveness of the implementing ADT. In general, the automatic synthesis of auxiliary functions not possible although inductive inference methods [Bar 83] may be useful. For example, in [JaT 87], an auxiliary operator is synthesised using such methods.

We illustrate the three possibilities described above by considering the implementation of the *add* operator from the **Queue** specification by the specification **SRelation2**, the SRelation derived from **Queue** and given as example 4.2 in §4.2.3. For simplicity of illustration, we omit the enrichment **Filter** in this example because the specification of the full implementation is very similar to the partial implementation. In **SRelation2**, we assume that the enriched specification **Nodes&Edges'** is substituted for the anarchic **Nodes&Edges** and that *add* is implemented by the operator *ADD*; i.e. **SRelation2** is enriched with the operator *ADD* and equation (*):

```
ADD : graph nat -> graph

∀t:queue,n:nat.
abs(ADD([⇓(t),⇒(t)],n))   = abs([⇓add(t,n),⇒add(t,n)])          (*)
```

5.6.3.1 An Example using the Equational Theory of the Implementing ADT

When we instantiate t in (*) by eq, we have

$\forall$n:nat.

abs(ADD([$\Downarrow$(eq),$\Rightarrow$(eq)],n)) = abs([$\Downarrow$add(eq,n),$\Rightarrow$add(eq,n)]) I1

Specifications for the first and second components of the r.h.s. of I1, using the restricted language of **SRelation2**, can be found in the equational theory of **SRelation2**:

$\Downarrow$add(eq,n) = front(add(eq,n)) $+1_n$ $\Downarrow$(dequeue(add(eq,n))) unfold B2

= n $+1_n$ $\Downarrow$eq unfoldR1,R4

$\Rightarrow$add(eq,n) = $[]_e$ unfold B5

= $\Rightarrow$eq fold Q6

And so by abs-dropping we have:

ADD([$\Downarrow$eq $\Rightarrow$eq],n) = [n $+1_n$ $\Downarrow$eq,$\Rightarrow$eq]. I2

5.6.3.2 An Example using the Inductive Theory of the Implementing ADT

When we instantiate t in (*) by add(q,n),we have

$\forall$q:queue,n,n1:nat.

abs(ADD([$\Downarrow$(add(q,n)),$\Rightarrow$(add(q,n))],n1))

= abs([$\Downarrow$add(add(q,n),n1),$\Rightarrow$add(add(q,n),n1)]) I3

Specifications for the first and second components of the r.h.s. of I3, using the restricted language of **SRelation2**, cannot be found in the equational theory of **SRelation2**. However, an implementation of the first component can be found in the

inductive theory of **SRelation**. We can apply the introduction rule to introduce an equation which is similar to the lemma given in example 3.14, §3.3, and proven in Appendix 3 as lemma 5, example 3.14. Namely, in order to synthesise an implementation of ⇓(add(q,n)) we need to show that

$$\forall q{:}queue, n{:}nat.\ \Downarrow(add(q,n)) = n +r_n \Downarrow q$$

is valid in the inductive theory of **SRelation2**.

We can prove the validity of this equation either by rewriting techniques or by explicit use of induction. Here, we choose the latter. In this, and subsequent proofs, we often apply more than one rule during each step; when the rules applied are only folds or unfolds, then we give only the relevant equation names as justification.

lemma: $\forall q{:}queue, n{:}nat.\ \Downarrow(add(q,n)) = n +r_n \Downarrow q$

base case:

⇓(add(eq,n))	$= n +l_n []_n$	R1,R4,Q1,Q2
	$= n +r_n []_n$	N4
	$= n +r_n \Downarrow eq$	Q3

induction step: Assume $\forall q{:}neq, n{:}nat.\ \Downarrow(add(q,n)) = n +r_n \Downarrow q$.

Consider ⇓(add(add(q,n),n1)).

⇓(add(add(q,n),n1))		
	$= front(add(add(q,n),n1)) +l_n \Downarrow(dequeue(add(add(q,n),n1)))$	Q2
	$= front(add(add(q,n),n1)) +l_n \Downarrow(add(dequeue(add(q,n)),n1)$	R5
	$= front(add(add(q,n),n1)) +l_n (n1 +r_n \Downarrow(dequeue(add(q,n))))$	ass.
	$= n1 +r_n (front(add(add(q,n),n1)) +l_n \Downarrow(dequeue(add(q,n))))$	N5
	$= n1 +r_n (front(add(q,n)) +l_n \Downarrow(dequeue(add(q,n))))$	R2
	$= n1 +r_n \Downarrow(add(q,n))$	Q2

Conclusion: $\forall q{:}queue, n{:}nat.\ \Downarrow(add(q,n)) = n +r_n \Downarrow q$.

And so by application of the introduction rule, we add the equation

$\forall q{:}queue, n{:}nat.\ \Downarrow(add(q,n)) = n +r_n \Downarrow q$ 14

5.6.3.3 An Example using the Inductive Theory of the Implementing ADT and Auxiliary Operations

Consider now an implementation for the second component of the r.h.s of I3. Again, we apply the introduction rule to introduce an equation which is similar to a lemma from example 3.14 and proven in Appendix 3 as lemmas 6, example 3.14. Namely, in order to synthesise an implementation of $\Rightarrow(add(q,n))$ we need to show that

$\forall q:queue, n, n1:nat.$

$$\Rightarrow(add(add(q,n),n1)) = \langle n,n1\rangle +r_e \Rightarrow(add(q,n)) \qquad \text{L1}$$

is valid in the inductive theory of **SRelation2**.

Unfortunately, this equation is not in the desired form; τ-sorted subterms of the l.h.s. occur on the r.h.s. We can avoid this situation by enriching the theory with a suitable auxiliary operation; i.e. we introduce the following operator:

$last: neq \rightarrow nat$

$\forall q:queue, n:nat.\ last(add(q,n)) = n \qquad \text{A5}$

Using equation A1, we can now rewrite L1 in the desired form as:

$\forall q:neq, n:nat.\ \Rightarrow(add(q,n)) = \langle last(q),n\rangle +r_e \Rightarrow q$

and we prove its validity by induction.

lemma: $\forall q:neq, n:nat.\ \Rightarrow(add(q,n)) = \langle last(q),n\rangle +r_e \Rightarrow q$

base case:

$\Rightarrow(add(add(eq,n),n1))$

$= \langle front(add(add(eq,n),n1)), front(dequeue(add(add(eq,n),n1)))\rangle$
$\quad +l_e \Rightarrow(dequeue(add(add(eq,n),n1))) \qquad \text{Q4}$

$= \langle n,n1\rangle +l_e \Rightarrow(add(eq,n1)) \qquad \text{R2,R5}$

$= \langle n,n1\rangle +l_e []_e \qquad \text{Q5}$

$= \langle last(add(eq,n)),n1\rangle +l_e []_e \qquad \text{A1}$

$= \langle last(add(eq,n)),n1\rangle +r_e []_e \qquad \text{E4}$

$= \langle last(add(eq,n)),n1\rangle +r_e \Rightarrow(add(eq,n)) \qquad \text{Q5}$

induction step: Assume $\forall q:neq, n:nat.\ \Rightarrow(add(q,n)) = \langle last(q),n\rangle +r_e \Rightarrow q.$

Consider $\Rightarrow(add(add(q,n),n1))$.

$\Rightarrow$ (add(add(q,n),n1))

= <front(add(add(q,n),n1)),front(dequeue(add(add(q,n),n1)))>
$+l_e$ $\Rightarrow$(dequeue(add(add(q,n),n1))) Q4

= <front(add(add(q,n),n1)),front(dequeue(add(add(q,n),n1)))>
$+l_e$ $\Rightarrow$(add(dequeue(add(q,n)),n1)) R5

= <front(add(add(q,n),n1)),front(dequeue(add(add(q,n),n1)))>
$+l_n$ (<last(dequeue(add(q,n))),n1>
$+r_e$ $\Rightarrow$(dequeue(add(q,n))) ass.

= <last(dequeue(add(q,n))),n1> $+r_e$
<front(add(add(q,n),n1)),front(dequeue(add(add(q,n),n1)))>
$+l_e$ $\Rightarrow$(dequeue(add(q,n))) E5

= <last(dequeue(add(q,n))),n1> $+r_e$
<front(add(q,n)),front(add(dequeue(add(q,n)),n1))>
$+l_e$ $\Rightarrow$(dequeue(add(q,n))) R5

= <last(dequeue(add(q,n))),n1> $+r_e$
<front(add(q,n)),front(dequeue(add(q,n)))>
$+l_e$ $\Rightarrow$(dequeue(add(q,n))) R2,R5

= <last(dequeue(add(q,n))),n1> $+r_e$ $\Rightarrow$(add(q,n)) Q4

= <last(add(q,n)),n1> $+r_e$ $\Rightarrow$(add(q,n)) R5,A1

Conclusion: $\forall$q:neq,n:nat. $\Rightarrow$(add(q,n)) = <last(q),n> $+r_e$ $\Rightarrow$q.

And so by application of the introduction rule we add the equation

$\forall$q:neq,n:nat. $\Rightarrow$(add(q,n)) = <last(q),n> $+r_e$ $\Rightarrow$q I6

By unfolding I3 with I4 and I6 we have

$\forall$q:neq,n:nat.

abs(ADD([$\Downarrow$q,$\Rightarrow$q],n))

= abs([n $+r_n$ $\Downarrow$q,<last(q),n> $+r_e$ $\Rightarrow$q]) I7

and by abs-dropping we have

$\forall$q:neq,n:nat.

ADD([$\Downarrow$q,$\Rightarrow$q],n)

= [n $+r_n$ $\Downarrow$q,<last(q),n> $+r_e$ $\Rightarrow$q] I8

Equations I2 and I8 together give an implementation for the *add* operator.

5.6.3.4 Synthesising Implementations using Inductive Inference

The crucial question to ask when synthesising an implementation which uses the inductive theory of the implementing ADT is which inductive theorem(s) are required? In this section we discuss how these theorems (introduced by so-called Eureka steps in [BuD 77]) can be deduced from *examples* of the implementation by exploiting the fact that every ground instance of an inductive theorem is valid in the equational theory of the ADT. The hope is that if we consider enough ground equations, then we may be able to find a generalisation which is an inductive theorem.

Such an approach has been taken in [Sum 75] where a method for synthesising Lisp programs from example computations is given. A more general theory of the synthesis of generalisations from examples is the aim of *inductive inference* [Bar 83]; [Lan 87] has given an inductive inference algorithm which, under some circumstances, is a generalisation of Summers' method and includes specifications with any signature.

An inductive inference algorithm generates a *hypothesis* from some examples; in our case, we want to consider enough examples of the implementation of an operator so that we can hypothesise a complete specification of the implementation. The algorithm of [Lan 87] requires that the specification be a classified specification with a compatible size measure (as described in §2.5.2) and it synthesises *loop programs* from a finite set of input/output examples. Informally, a loop program in this context is a recursion equation in the form

$$f(x) = ...f(s(x))...$$

where s is a (*Lange*) selector.

We briefly illustrate the algorithm, and how we would use it, with the synthesis of the implementation of $\Rightarrow$`add(q,0)`; we do not consider the synthesis of the implementation of $\Rightarrow$`add(q,n)` for simplicity and because the algorithm given in [Lan 87] considers functions with arity 1. In order to apply the algorithm, we have to classify the **SRelation2** specification and define a compatible size measure. In this example, we assume that selectors are (Lange) selectors, the size measure is based on the size of ground generator terms (instead of the **leaves** function), and $[]_e$ and $+r_e$ are the constructors; the motivation for classifying $+r_e$ as a constructor instead of the generator $+l_e$ will be given later in the example.

In the example, we aim to synthesise a loop program for the ADD0 operation where ADD0 is specified by

```
        ADD0 : ecol -> ecol
(∀Q:ecol) abs(ADD0(Q)) = add(abs(Q),0).
```

Assume, in the following, that we also have the partial specification of the implementation of the operator last on sort ecol; i.e. LAST : ecol -> nat. (Strictly speaking, an operator with this arity is not allowed. We permit it in this example because, for simplicity, we are not considering implementations on sort graph, but on sort ecol.)

The input/output examples are pairs of ground constructor terms $<t_1, t_2>$ s.t. $ADD0(t_1) \equiv_E t_2$. We are only interested in the specification of ADD0 when applied to representations of the sort *queue* and so we give the following examples:

1) $<\Rightarrow eq, []_e>$

2) $<\Rightarrow add(eq,1), <1,0> +r_e []_e>$

3) $<\Rightarrow add(add(eq,2),3), <3,0> +r_e (<2,3> +r_e []_e)>$

which are found by substituting eq, add(eq,1), and add(add(eq,2),3) resp. for Q and applying the fold and unfold rules.

The algorithm consists of three steps. The first step is to express every output in terms of the input; that is, given example $<X_i, Y_i>$, compute a term t_i from $T_{\Sigma c}(T_{\Sigma se}(\{x\}))$ such that $t_i [x / X_i] \equiv_E Y_i$. We note that each computed term t_i is not necessarily a unique solution. For the given examples, we find the following terms:

$t_1 =_{def} []_e$

$t_2 =_{def} <LAST(x), 0> +r_e []_e$

$t_3 =_{def} <LAST(x), 0> +r_e (<2,3> +r_e []_e)$

We note that in this example, the operator ADD0 does not occur in the computed terms.

The second step is to find the common computational paths in the examples; namely, for each pair of terms t_i and t_j found in the first step, we find a generalisation t from $T_{\Sigma c}(T_{\Sigma se}(\{x\}))$ such that $t[x / X_i] \equiv_E t_i[x / X_i]$ and $t[x / X_j] \equiv_E t_j[x / X_j]$. For example, a generalisation of t_2 and t_3 is $\langle last(x), 0\rangle +r_e x$, whereas a generalisation of t_1 and t_2 or t_1 and t_3 cannot be found. Obviously, the search for a generalisation must be controlled. In general, the existence of a generalisation is undecidable; [Lan 87] has proven that it is decidable when the specification is classified and has a compatible size measure. The restriction of the generalisation to terms in $T_{\Sigma c}(T_{\Sigma se}(\{x\}))$ ensures that loop programs can be synthesised; had we chosen $+1_e$ as the constructor then generalisations could not have been found without enriching the specification.

The final step combines the generalisations into a set of equations specifying ADD; for this example, we have

$$ADD0([]_e) = []_e$$

$$\forall e{:}pair, E{:}ecol.\ ADD0(e +r_e E) = \langle LAST(E), 0\rangle +r_e (e +r_e E).$$

We note that this specification specifies ADD0 even when applied to elements of sort *ecol* which are not representations of the sort *queue*. Also, because ADD0 does not appear in the terms found in the first step, the equations are not recursive. We could easily synthesise ADD: ecol nat -> ecol either by applying the algorithm again, or by extending the algorithm to take tuples of inputs in the first step.

There may be problems with this approach. Although it is easy to see that the above three examples are sufficient to derive a hypothesis which is a complete specification of ADD; in general, the problem of determining the number and nature of the examples is undecidable. Moreover, we may not be able to find generalisations at all without enriching the specification with auxiliary operators. The problem of determining which auxiliary operations are required is difficult. If we have a candidate generalisation in mind, then we may be able to use inductive inference methods to synthesise the required auxiliary operator as illustrated in [JaT 87]. Often, auxiliary operators are unnecessary if the existing constructors have permutative properties. For example, the operator $+r_e$ would not have been required if $+1_e$ had been commutative. We hope that in future, the inductive inference algorithm will be extended to allow those permutative properties which have an associated unification procedure (for example, the

associative-commutative property).

5.6.4 Partial Implementation Examples

We conclude this section with some example (partial) implementations and the access operators which are the result. The particular examples are included for two reasons: either they are implementations of familiar specifications, or they illustrate an interesting aspect of the implementation, (in the constrained form), of a specification by its storage relations. The implementations are given as enrichments of the respective storage relation specification (from §4.3); for simplicity, we assume that the implementation of each operator is given by an operator with the same name in upper case.

Example 5.8: IMPS: a partial specification of **Stack/SRelation1.**

```
spec    IMPS = enrich SRelation1 + Filter by

opns    CREATE  : rgraph
        PUSH    : rgraph nat -> rgraph
        POP     : rgraph -> rgraph
        TOP     : rgraph -> nat
        EMPTY   : rgraph -> bool

eqns
CREATE                  = [[]n,[]e]
PUSH(rep(create),n)     = [n +1n ⇓create,⇒create]          if ok(create)
∀s:nes,n:nat.
PUSH(rep(s),n)          = [n +1n⇓s,<n,top(s)> +1n ⇒s]      if ok(s)

POP(rep(create))        = [⇓create,⇒create]                if ok(create)
∀n:nat.
POP(rep(push(create,n)))
  = [⇓push(create,n) \n top(push(create,n)),⇒create]       if ok(create)
∀s:nes,n,n1:nat.
POP(rep(push(s,n)))
  = [⇓push(s,n) \n top(push(s,n)),
     ⇒push(s,n) \\e scen(⇒push(s,n),top(push(s,n)))]      if ok(s)

∀s:nes.
TOP(rep(s))             = top(s)                           if ok(s)

∀s:nes.
EMPTY(rep(s))           = F                                if ok(s)
EMPTY(rep(create))      = T
end
```

Example 5.9: IMPQ: a partial specification of **Queue/SRelation2.**

```
spec    IMPQ = enrich SRelation2 + Filter by

opns    EQ      : rgraph
        ADD     : rgraph nat -> rgraph
        DEQUEUE : rgraph -> rgraph
        FRONT   : rgraph -> nat
        last    : neq -> nat
        EMPTY   : rgraph -> bool

eqns
EQ                      = [[]n, []e]

ADD(rep(eq))            = [n +rn ⇓eq, ⇒eq]
∀q:neq, n:nat.
ADD(rep(q))             = [n +rn⇓q, <last(q), n> +rn ⇒q]      if ok(q)

DEQUEUE(rep(eq))        = [⇓eq, ⇒eq]
∀n:nat.
DEQUEUE(rep(add(eq,n)))
= [⇓add(eq,n) \n front(add(eq,n)), ⇒eq]
∀q:neq, n, n1:nat.
DEQUEUE(rep(add(q,n)))
   = [⇓add(q,n) \n front(add(q,n),
     ⇒add(q,n) \\e (scen(⇒add(q,n), front(q,n)))      if ok(add(q,n))

∀q:neq.
FRONT (rep(q))          = top(q)                          if ok(q)

∀q:neq.
EMPTY(rep(q))           = F                               if ok(q)
EMPTY(rep(eq))          = T
end
```

In this example, the full implementation would have to more expressive. Informally, the operations named by $\backslash_n$, $\backslash_e$, $\backslash\backslash_n$, and $\backslash\backslash_e$ "work" from left to right; i.e. `N` $\backslash_n$ `x` removes the leftmost occurrence of `x` from (the ground generator form of) `N`. If there were several occurrences of `x` in `N`, as there might be in a full implementation, then an operation which removes the rightmost occurrence of `x` would be required; namely, the fifth equation, as given above, is not valid for all instantiations of `q`.

Example 5.10: IMPT: a partial specification of **Binary_Tree_2 /SRelation4.**

spec **IMPT = *enrich* SRelation4 + Filter *by***

opns

M	:	nat -> rgraph
COMB	:	rgraph rgraph nat -> rgraph
ROOT	:	rgraph -> nat
LEFT	:	rgraph -> nat
RIGHT	:	rgraph -> nat
LEAF	:	rgraph -> bool
$child_n$	:	ncol ecol -> ncol
$child_e$	:	ncol ecol -> ecol
rright	:	nonleaf -> nat
rleft	:	nonleaf -> nat

eqns

$\forall n$:nat.
$M(n) = [n +1_n []_n, []_e]$

$\forall t,s$:tree, n:nat.
$COMB(rep(t),rep(s)) = [n +1_n (\Downarrow t\ plus_n \Downarrow s),\ (<n,root(t)> +1_e <n,root(s)>)\ plus_e\ (\Rightarrow t\ plus_e \Rightarrow s)]$ if $(ok(t) \wedge ok(s))$

$\forall t$:tree.
$ROOT(rep(t)) = root(t)$ if $ok(t)$

$\forall t$:nonleaf.
$LEFT(rep(t)) = [(\Downarrow t \setminus_n root(t)) \setminus\setminus_n (child_n(rright(t)) +1_n []_n),\ (\Rightarrow t \setminus_e <root(t),rright(t)>) \setminus\setminus_e (child_n(rright(t)) +1_e []_e)]$ if $ok(t)$

$\forall t$:nonleaf.
$RIGHT(rep(t)) = [(\Downarrow t \setminus_n root(t)) \setminus\setminus_n (child_n(rleft(t)) +1_n []_n),\ (\Rightarrow t \setminus_e <root(t),rleft(t)>) \setminus\setminus_e (child_n(rleft(t)) +1_e []_e)]$ if $ok(t)$

$\forall N$:ncol, E:ecol.
$child_n(n +1_n N,E) = n +1_n (child_n(adj(E,n),E)\ plus_n\ child_n(N,E))$

$child_n([]_n,E) = []_n$

$\forall N$:ncol, E:ecol.
$child_e(n +1_n N,E) = scen(E,n)\ plus_e\ (child_e(adj(E,n),E)\ plus_e\ child_e(N,E))$
$child_e([]_n,E) = []_e$

```
∀t:nonleaf.
rright(t)          = root(right(t))

∀t:nonleaf.
rright(t)          = root(right(t))

∀t:nonleaf.
LEAF(rep(t))       = F
∀n:nat.
LEAF(M(n))         = T
end
```

We note that the partial implementation of **Binary_Tree** is considerably more complex than this example because although when $ok(t) \equiv_E T$, there is an injection from the leaves of t to the contents set of t, it is not surjective. Thus, it is still possible that "positions" in the storage relations may be identified which are not derived in equivalent ways.

For example, if t is comb(comb(m(1),m(2)),comb(m(0),m(3))) then we have the situation that [root(right(t))] = [root(left(t))] but [right(t)] ≠ [left(t)].

Example 5.11: IMPR: a partial specification of **Reversible_List_1/SRelation5.**

```
spec    IMPR = enrich SRelation5 + Filter by

opns    NIL     : rgraph
        _:_     : nat rgraph -> rgraph
        TL      : rgraph -> rgraph
        HD      : rgraph -> nat
        hdrev   : nel -> nat
        REV     : rgraph -> rgraph
        EMPTY   : rgraph -> bool

eqns
NIL             = [[]n,[]e]

∀n:nat.
n:rep(nil)      = [n +1n ⇓nil,⇒nil]

∀l:nel,n:nat.
n:rep(l)
  = [n +1n⇓l,<n,hd(l)> +1e (<hd(l),n> +1e ⇒l)]                    if ok(l)

TL(rep(nil))    = [[]n,[]e]
```

```
∀n:nat.
TL(rep(n:nil))      = [[]n, []e]

∀l:nel,n:nat.
TL(rep(n:l))
  = [⇓(n:l)  \n hd(n:l),
     (⇒(n:l) \\e scen(⇒(n:l),hd(n:l)))  \\e tgtn(⇒(n:l),hd(n:l))]
                                                        if ok(n:l)

REV(rep(nil))               = [[]n, []e]

∀n:nat.
REV(rep(n:nil))
  = [hdrev(n:nil) +1n (⇓(n:l)  \n hdrev(n:l)),
     <hdrev(n:l),hd(n:nil)> +1e
          (⇒(n:nil)  \e <hdrev(n:l),hd(n:nil)>)]

∀l:nel,n:nat.
REV(rep(n:l))
  = [hdrev(n:l) +1n (⇓(n:l)  \n hdrev(n:l)),
     hd(scen(⇒(n:l),hdrev(n:l)) +1e  hd(tgtn(⇒(n:l),hdrev(n:l)))
      +1e REV([⇓(n:l)  \n hdrev(n:l),
                (⇒(n:l)  \\e scen(⇒(n:l),hdrev(n:l) )))
                \\e tgtn(⇒ (n:l),hdrev(n:l)))]          if ok(n:l)

∀l:nel.
HD(rep(l))          = hd(l)                                  if ok(l)

∀l:nel.
hdrev(l)            = hd(rev(l))                             if ok(l)

∀l:nel.
EMPTY(rep(l))       = F                                      if ok(l)
EMPTY(rep(nil))     = T
end
```

We note that this is the first example which uses recursion when specifying the implemented operators; i.e. the implementation of `rev`, `REV`, is recursive. Moreover, we note that the effect of the `REV` operator, when we look at the ground instances, is to permute the subterms of the argument. Namely, the order in which the "elements" of the `ncol` and `ecol` terms are presented is changed.

Example 5.12: IMPL1: a partial specification of **Circular_List_right/SRelation6.**

spec **IMPL1 = *enrich* SRelation6 + Filter *by***

opns

```
NIL      : rgraph
_:_      : nat rgraph -> rgraph
TL       : rgraph -> rgraph
HD       : rgraph -> nat
hdshift  : nel -> nat
last     : nel -> nat
SHIFT    : rgraph -> rgraph
EMPTY    : rgraph -> bool
```

eqns

$NIL = [[]_n, []_e]$

$\forall n{:}nat.$
$n{:}rep(nil) = [n +1_n \Downarrow nil, \Rightarrow nil]$

$\forall l{:}nel, n{:}nat.$
$n{:}rep(l) = [n +1_n \Downarrow l, <n, hd(l)> +1_e (<hd(l), n> +1_e \Rightarrow l)]$ if ok(l)

$TL(rep(nil)) = [[]_n, []_e]$

$\forall n{:}nat.$
$TL(rep(n{:}nil)) = [n +1_n []_n, []_e]$

$\forall l{:}nel, n{:}nat.$
$TL(rep(n{:}l))$
$= [(\Downarrow(n{:}l) \setminus_n hd(n{:}l)) plus_n hd(n;l),$
$((\Rightarrow(n{:}l) \setminus\setminus_e scen(\Rightarrow(n{:}l), hd(n{:}l))) \setminus\setminus_e scen(\Rightarrow(n{:}l), last(n{:}l))$
$plus_e (<last(l), hdshift(l)> 1_e []_e)]$ if ok(n:l)

$\forall n{:}nat.$
$last(n{:}nil) = n$
$\forall l{:}nel, n{:}nat.$
$last(n{:}l) = last(l)$

$\forall l{:}nel.$
$hdshift(l) = hd(shift(l))$

$SHIFT(rep(nil)) = [[]_n, []_e]$
$\forall n{:}nat, l{:}nel.$
$SHIFT(rep(n{:}l)) = [\Downarrow(n{:}l), \Rightarrow(n{:}l)]$

```
∀l:nel,n:nat.
SHIFT(rep(n:l))
  = [(⇓(n:l) \n hd(n:l)) plus_n hd(n:l),
     (((⇒(n:l) \\e scen(⇒(n:l),hd(n:l)))
        \\e scen(⇒(n:l),last(n:l))) plus_e
        (scen(⇒(n:l),hd(n:l)) plus_e scen(⇒(n:l),last(n:l))))]
                                                    if ok(n:l)
∀l:nel.
HD(rep(l))          = hd(l)                          if ok(l)

∀l:nel.
EMPTY(rep(l))       = F                              if ok(l)
EMPTY(rep(nil))     = T
end
```

Example 5.13: IMPL2: a partial specification of **Circular_List_2/SRelation7.**

```
spec    IMPL2 = enrich SRelation7 + Filter by

opns    NIL      : rgraph
        _:_      : nat rgraph -> rgraph
        TL       : rgraph -> rgraph
        HD       : rgraph -> nat
        hdshift  : nel -> nat
        next     : nel -> nat
        SHIFT    : rgraph -> rgraph
        EMPTY    : rgraph -> bool

eqns
NIL             = [[]_n,[]_e]

∀n:nat.
n:rep(nil)      = [n +l_n ⇓nil,⇒nil]

∀l:nel,n:nat.
n:rep(l)
  = [n +l_n ⇓l,
     (⇒l \\e scen(⇒l,hdshift(l)) plus_e
       (<n,hd(l)> +l_e (<hdshift(l),n> +l_e []_e))]          if ok(l)

TL(rep(nil))    = [[]_n,[]_e]

∀n:nat.
TL(rep(n:nil))  = [n +l_e []_n,[]_e]
```

```
∀l:nel,n:nat.
TL(rep(n:l))
  = [⇓(n:l) \n hd(n:l),
    <hdshift(n:l),next(n:l_)> +1e
     ((⇒(n:l) \e <hd(n:l),hd(n:l)>) \e <hdshift(n:l),hd(n:l)>)]
                                                    if ok(n:l)
next(n:nil)                 = n

∀l:nel.
next(n:l)                   = hd(tl(n:l))

∀l:nel.
hdshift(l)                  = hd(shift(l))

SHIFT(rep(nil))             = [[]n,[]e]

∀n:nat.
SHIFT(rep(n:nil))           = [n +1n []n,[]e]

∀l:nel,n:nat.
SHIFT(rep(n:l))
  = [hdshift(n:l) +1n (⇓(n:l) \n hdshift(n:l)),
     scen(⇒(n:l),hdshift(n:l)) plus e
          (⇒(n:l) \\e scen(⇒(n:l),hdshift(n:l)))]     if ok(n:l)


∀l:nel.
HD(rep(l))          = hd(l)                                 if ok(l)

∀l:nel.
EMPTY(rep(l))       = F                                     if ok(l)
EMPTY(rep(nil))     = T
end
```

It is interesting to compare these last two (partial) implementations. Both object specifications have the same storage type, and clearly *last* and *hdshift* in **IMPL1** specify the same behaviour as *hdshift* and *next* resp. in **IMPL2**. The difference is in the way in which they are specified: *last* is defined recursively whereas *hdshift*, (in both cases), is defined by the composition of a selector and rearranger and *next* is defined by the composition of a selector and eliminator. We note that in both cases, the effect of the *SHIFT* operator, (when we look at the ground instances), is to permute the subterms of the argument.

To conclude this section, we give the access operators which result from the example (partial) implementations.

Example 5.8: `Stack`
The access operators are {`top`}.

Example 5.9: `Queue`
The access operators are {`front, last`}.

Example 5.10: `Binary_Tree_2`
The access operators are {`root, rleft, rright`}.

Example 5.11: `Reversible_List_1`
The access operators are {`hd, hdrev`}.

Example 5.12: `Circular_List_right`
The access operators are {`hd, hdshift, last`}.

Example 5.13: `Circular_List_left`
The access operators are {`hd, hdshift, next`}.

5.7. Comparison of Methods I and II and Optimisations

The second method of determining the access operators is generally preferable because it includes an element of efficiency analysis. This method is perhaps similar to Scherlis' idea of *specialisation* [Sch 85]: here, we synthesise new selectors within the context of implementation by storage relation. When the specifications of those operations consist of compositions of existing selectors and operators, then they describe the specialistion of the given selectors to the implementation.

Unlike the first method, the second method is not an "automatic" strategy and this aspect has its drawbacks.

First, in general, the introduction of the appropriate auxiliary operators is difficult. The introduction of selectors may be relatively straightforward. For example, for each occurrence of $s(f_1(f_2(\ldots(t)\ldots)))$, where s is a selector, we can always introduce an auxiliary operator s' such that $s'(t) =_{def} s(f_1(f_2(\ldots(t)\ldots)))$..

Second, a finite presentation of the (partial) implementation of implicitly keyed ADTs in the constrained form may not be possible. For example, in order to give a (partial) implementation of the `add` operator in the **`Priority_Queue`** specification (cf. §2.4.1), we need to specify an operator which returns the appropriate position in the (linear) storage relation where insertion should occur. This operation cannot be specified without violating the constraints; for example it is specified either by a (disallowed) operation with arity `pq nat -> nat` and behaviour "return the nth position", or by infinitely many selectors returning the first, second, third, and so on, positions.

Both strategies may be optimised according to properties of the given object specification. For example, both methods may include, as an access node, a node which is directly reachable from another access node. This allocation of access nodes may constitute an unwise trade-off between space for time. Such a situation occurs in several of our examples: when considering **`Queue`**, we can show that for all `q:neq`, `last(q)` $\Rightarrow_q$ `front(q)`, when considering **`Binary_Tree_2`**, we can show that for all `t:tree`, `root(t)` $\Rightarrow_t$ `rright(t)`, when considering **`Circular_List_left`**, we can show that for all `l:nel`, `hd(l)` $\Rightarrow_l$ `next(l)` and `hdshift(l)` $\Rightarrow_l$ `hd(l)`, and when considering **`Circular_List_right`**, we can show the that for all `l:nel`, `hd(l)` $\Rightarrow_l$ `hdshift(l)` and `last(l)` $\Rightarrow_l$ `hd(l)`.

For a given storage relation $\Rightarrow_t$, we define the *distance* between two primitive sorted elements in the reflexive, transitive closure of $\Rightarrow_t$ in the following way:

> **Definition:** Let $\Rightarrow_t$ be a storage relation and let $\Rightarrow_t^*$ be its reflexive, transitive closure. For distinct primitive sorted elements a, b with $a \Rightarrow_t^* b$, we define the ***distance***, d(a,b), between a and b as the minimum number n+1, $n \geq 0$, of intermediate elements x_i, $1 \leq i \leq n$, such that
>
> $$a \Rightarrow_t x_1,\ x_1 \Rightarrow_t x_2,\ \ldots,\ x_n \Rightarrow_t b.$$
>
> When $a \equiv b$ then we define d(a,b) to be 0; d(a,b) is undefined for a,b not related by $\Rightarrow_t^*$.

One way in which we may optimise a set of access nodes, or sets of access nodes, is to remove (some of) the access nodes which are less than distance k, for some k, from another access node. When A is a set of access nodes, we say that a subset O of A is a *k-optimised* set when $(\forall x \in A \backslash O)(\exists y \in O: d(y,x) \leq k)$. We note that in a given set of access nodes, there may be several nodes which are candidates for removal.

For example, consider the **Circular_List_left** specification and $\Rightarrow$(4:3:2:1:nil).{4,3,1} is the set of (labels of) access nodes determined by the second method; {1,3} and {1,4} are 1-optimised sets and {1} is a 2-optimised set. This optimisation generalises to the exclusion of access operators. Thus, because for all l:nel and n:nat.{next(n:l),hdshift(n:l)} and {hd(n:l),hdshift(n:l)} are sets of (the labels of) 1-optimised access nodes of $\Rightarrow$n:l, we refer to {next,hdshift} and {hd,hdshift} as 1-optimised sets of access operators.

We note that in the present context, we cannot replace any of the access operators (in the implementations) with non-selectors or compositions thereof because all access to the "data" in the elements of sort ncol and ecol is constrained to be explicitly keyed.

5.8 Specifying Storage Graphs

Now we extend our representation of the derived sort: we consider the storage relation at t and (the labels of) its access nodes as the representation of t. Namely, each element of the derived sort will be represented by a tuple consisting of the contents set, the storage relation, and the (labels of the) access nodes of the storage relation.

Informally, we need to enrich each **SRelation** specification with a new sort sgraph, its subsort rsgraph, an sgraph constructor, new abs and rep operators with arity τ -> rsgraph and rsgraph -> τ resp., and the appropriate equations specifying rep; i.e. when $s_1, \ldots, s_n$ are the access operators, then we add the equation

$$\text{rep}(t) = [\Downarrow t, \Rightarrow t, s_1(t), \ldots, s_n(t)].$$

The access operators may not be defined for all of the domain of rep; moreover, they may all have different arities. So, either we have to introduce further subsorts into sgraph and overload the sgraph constructor with several arities, or we have to enrich the (representation of the) primitive sort with an additional element, null, say, to

represent the undefined value. We adopt the latter approach because it is simple and is closest to our intuition about what happens in pointer implementations. For example, in the **Queue** example we might give the `sgraph` constructor by `[_,_,_,_]:ncol ecol nat nat -> sgraph`, and the representation equations by:

$$\forall q{:}neq.\quad rep(q) = [\Downarrow q, \Rightarrow q, front(q), last(q)]$$
$$rep(eq) = [\Downarrow eq, \Rightarrow eq, null, null].$$

To summarise, informally, from each **SRelation** specification derived from object specification **Spec**, we derive an **SGraph** specification in the following way. Let $\{s_1, \ldots, s_n\}$, n>0, be the set of the access operators which define the access nodes of the storage relations of **Spec**. Of course each element of $\{s_1, \ldots, s_n\}$ need not be applicable to all of the derived sort elements of **Spec**. We enrich **SRelation** with the sort `sgraph`, its subsort `rsgraph`, and the constant `null`:δ; then we change the arities of `rep` and `abs` to τ -> `rsgraph` and `rsgraph` -> τ resp. We keep the equations relating `abs` and `rep` (i.e. `abs(rep(t)) = t`), but we replace the equations relating the operators `rep` and the graph constructor by equations of the form

$$rep(t) = [\Downarrow t, \Rightarrow t, x_1(t), \ldots, x_n(t)]$$

where for all i, $1 \leq i \leq n$, $x_i(t)$ is `null` when $s_i(t)$ is not well-typed and $s_i(t)$ otherwise.

In addition, we now relax the constraint that access to the "data" in the elements of sort `ncol` and `ecol` is explicitly keyed; we introduce the following selectors hd_n and hd_e and the eliminators tl_n and tl_e into every **SGraph** specification. Again, the sorts are described with an abuse of notation.

hd_n : ncol -> δ

hd_e : ecol -> pair

tl_n : ncol -> ncol

tl_e : ecol -> ecol

$\forall n{:}\delta, N{:}ncol.\quad hd_n(n +1_n N) = n$

$\forall e{:}pair, E{:}ecol.\quad hd_e(e +1_e E) = e$

$\forall n{:}\delta, N{:}ncol.\quad tl_n(n +1_n N) = N$

$\forall e{:}pair, E{:}ecol.\quad tl_e(e +1_e E) = E$

The introduction of these operators will allow us to optimise the access operators (if required) and to consider implementing object operations by storage graphs without introducing further synthesised selectors.

5.9 Implementing Object Specifications by Storage Graphs

Finally, we can consider the aim of this chapter: the synthesis of **Spec/SGraph**.

Most of the work of constructing **Spec/SGraph** has already been carried out whilst constructing **Spec/SRelation**. For example, when we have

$$\texttt{rep(t)} \quad = \quad [\Downarrow \texttt{t}, \Rightarrow \texttt{t}, \texttt{s}_1\texttt{(t)}, \ldots, \texttt{s}_n\texttt{(t)}]$$

then for every operator `f: τ -> τ`, the representation of `f`, `F: sgraph -> sgraph` is defined by

$$\texttt{F(rep(t))} \quad = \quad [\Downarrow \texttt{f(t)}, \Rightarrow \texttt{f(t)}, \texttt{s}_1\texttt{(f(t))}, \ldots, \texttt{s}_n\texttt{(f(t))}]$$

Namely, the specifications of the first two components are given in the specification of **Spec/SRelation**; we have only to synthesise specifications for the the effect of the object operations on the access nodes. For example, we need to synthesise specifications for the composition of the access operators with f. (Of course, such a composition may not be well-typed in which case the result is `null`). The synthesis of the composition could involve the synthesis of more auxiliary operators. If any of these operators are selectors, then we would have to define a new representation and implementation which might, in turn, require further synthesised selectors etc. In order to avoid such a potentially infinite process, we decide to interate only once and we therefore forbid the synthesis of selectors during the implementation of the object specification by its storage graphs. We can enforce this constraint and still be ensured of an implementation because the enriched theories of **Ncol** and **Ecol** are now lists.

We conclude this section with some examples of (parts of) the storage graph implementations of the examples discussed in §5.6.4.

5.9.1 Examples from Storage Graph Implementations

For the sake of brevity, we do not include the entire specifications. Instead, for each example implementation we give

i) the arity of the `sgraph` constructor,

ii) the equations for rep,

iii) the specification of each derived sort operator composed with each access operator,

iv) if the access nodes have been optimised, then we also give specifications (in terms of the **SRelation** language) of the removed operators.

Example 5.14: from the storage graph implementation of **Stack**.

```
ops
[_,_,_]: ncol ecol nat -> sgraph

eqns
∀s:nes.          rep(s)                  = [⇓s, ⇒s, top(s)]
                 rep(create)             = [⇓create, ⇒create, null]

∀s:nes,n:nat.    top(pop(push(s,n)))     = hd(adj(⇒(push(s,n)),n))
end
```

Example 5.15: from the storage graph implementation of **Queue**.

```
ops
[_,_,_,_]: ncol ecol nat nat -> sgraph

eqns
∀q:neq.          rep(q)                  = [⇓q, ⇒q, front(q), last(q)]
                 rep(eq)                 = [⇓eq, ⇒eq, null, null]

∀q:neq,n:nat.    front(add(q,n))         = front(q)
∀q:neq,n:nat.    last(add(q,n))          = n

∀q:neq,n:nat.    front(dequeue(add(q,n))) =
                          hd(adj(⇒(add(q,n)),front(add(q,n))))
∀q:neq,n:nat.    last(dequeue(add(q,n)))  = last(q)
end
```

Example 5.16: from the storage graph implementation of **Queue** with 1-optimised access nodes.

```
ops
[_,_,_,_]: ncol ecol nat -> sgraph

eqns
∀q:neq.          rep(q)                    = [⇓q, ⇒q, last(q)]
                 rep(eq)                   = [⇓eq, ⇒eq, null]

∀q:neq,n:nat.    last(dequeue(add(q,n)))   = last(q)
∀q:neq,n:nat.    last(add(q,n))            = n

∀,n:nat.         front(add(eq,n))          = n
∀q:neq,n:nat.    front(add(q,n))           =
                          hd(adj(⇒(add(q,n)),last(add(q,n))))

end
```

Example 5.17: from the storage graph implementation of **Binary_Tree_2** with 1-optimised access nodes.

```
ops
[_,_,_,]: ncol ecol nat -> sgraph

eqns
∀t:tree.              rep(t)               = [⇓t, ⇒t, root(t)]

∀t,t':tree,n:nat.     root(comb(t,t',n))   = n
∀n:nat.               root(m(n))           = n

∀t:nonleaf.           root(left(t))        = hd(adj(⇒t,root(t)))
∀t:nonleaf.           root(right(t))       = hd(tl(adj(⇒t,root(t))))

∀t:nonleaf.           rright(t)            = hd(tl(adj(⇒t,root(t))))
∀t:nonleaf.           rleft(t)             = hd(adj(⇒t,root(t)))
end
```

Example 5.18: from the storage graph implementation of **Reversible_List_1**.

```
ops
[_,_,_,]: ncol ecol nat nat -> sgraph

eqns
∀l:nel.               rep(l)      = [⇓l, ⇒l, hd(l), hdrev(l)]
                      rep(nil)    = [⇓nil, ⇒nil, null, null]

∀n:nat,l:list.        hd(n:l)          = n
∀l:nel,n:nat.         hdrev(n:l)       = hdrev(l)
∀n:nat.               hdrev(n:nil)     = n

∀n:nat,l:nel.         hd(tl(n:l))      = hd(adj(⇒(n:l),hd(n:l)))
```

```
∀n:nat,l:nel.       hdrev(tl(n:l)) = hdrev(n:l)

∀n:nat.             hd(rev(n:nil))      = hd(n:nil)
∀n:nat,l:nel.       hd(rev(n:l))        = hdrev(n:l)
∀n:nat,l:nel.       hdrev(rev(n:l))     = hd(n:l)
∀n:nat.             hdrev(rev(n:nil))   = hdrev(n:nil)
end
```

Example 5.19: from the storage graph implementation of **`Circular_List_left`**.

```
ops
[_,_,_]: ncol ecol nat nat nat -> sgraph

eqns
∀l:nel.    rep(l)   = [⇓l, ⇒l, hd(l), next(l), hdrev(l)]
           rep(nil) = [⇓nil, ⇒nil, null, null, null]

∀l:list,n:nat.      hd(n:l)         = n
∀l:list,n:nat.      next(n:l)       = hdrev(l)
∀l:list,n:nat.      hdshift(n:nil) = hdshift(l)

∀n:nat,l:nel.       hd(tl(n:l))         = hd(adj(⇒(n:l),hd(n:l)))
∀n:nat,l:nel.       next(tl(n:l))       = hd(adj(⇒(n:l),next(n:l)))
∀n:nat,l:nel.       hdshift(tl(n:l))    = hdshift(n:l)

∀n:nat,l:list.      hd(shift(n:l))      = hdshift(n:l)
∀n:nat,l:list.      next(shift(n:l))    = hd(n:l)
∀n:nat.             hdshift(shift(n:nil))   = n
∀n:nat,l:nel.       hdshift(shift(n:l))     =
                              hd(tgtn(⇒(n:l),hdshift(n:l)))
end
```

Example 5.20: from the storage graph implementation of **`Circular_List_right`** with 1-optimised access nodes.

```
ops
[_,_,_]: ncol ecol nat nat -> sgraph

eqns
∀l:nel.    rep(l)   = [⇓l, ⇒l, hd(l), last(l)]
           rep(nil) = [⇓nil, ⇒nil, null, null]


∀l:list,n:nat.      hd(n:l)             = n
∀l:nel,n:nat.       last(n:l)           = last(l)
∀n:nat.             last(n:nil)         = n

∀n:nat,l:nel.       hd(tl(n:l))         = hd(adj(⇒(n:l),hd(n:l)))
∀n:nat,l:nel.       last(tl(n:l))       = last(n:l)
```

```
∀n:nat,l:nel.      hd(shift(n:l))      = hd(adj(⇒(n:l),hd(n:l)))
∀n:nat.            hd(shift(n:nil))    = n

∀n:nat,l:list.     hd(shift(n:l))      = hd(adj(⇒(n:l),hd(n:l)))
∀n:nat,l:list.     last(shift(n:l))    = hd(n:l)

∀n:nat,l:nel.      hdshift(n:l))       = hd(adj(⇒(n:l),hd(n:l)))
∀n:nat.            hdshift(n:nil)      = n
end
```

It is important to note that sometimes in **SGraph/SRelation**, the expressions for ⇓f(t) and ⇒f(t) taken from **Spec/SRelation** can be replaced by much simpler expressions. This may be possible because the information given by the access nodes gives us an additional way of distinguishing between the storage graphs of discrete derived elements; we may therefore be able to simplify the `ncol` and `ecol` components of the **SGraph** implementation. Essentially, we can have the situation where given `t` and `t'`, derived sort terms in different congruence classes, and selector `s` such that the images of `t` and `t'` under `s` are in different congruence classes, the `t` may be represented by [⇓t, ⇓t, s(t)] and `t'` by [⇓t, ⇓t, s(t')]. The representations preserve the distinction between `t` and `t'` because `s(t)` and `s(t')` are distinct.

For example, the implementation of the *shift* operator in the **Circular_List_right** specification, `SHIFT: rsgraph -> rsgraph`, is given by:

```
SHIFT(rep(nil))     = [⇓nil, ⇒nil, null, null]
∀n:nat.
SHIFT(rep(n:nil)) = [⇓(n:nil), ⇒(n:nil), n, n]
∀n:nat.l:nel.
SHIFT(rep(n:l))
          = [⇓(n:l), ⇒(n:l), hd(adj(⇒(n:l),hd(n:l)), hd(n:l)]
```

It is easy to see that the *ncol* and *ecol* components of this specification are much simpler than the specification of *SHIFT* given in Example 5.12. Simplifications such as these are important: they can lead to more efficient imperative implementations. Moreover, the simplifications may even allow for the removal of recursion in the *ncol* and *ecol* components. For example, the implementation of the *rev* operator in the **Reversible_List_1** specification, `REV: rgraph -> rgraph`, is specified by a recursion equation in Example 5.11. However, when we consider the *sgraph* implementation, we can remove the recursion entirely and keep the first two components

of the range of REV identical to the first two components of the domain; i.e. the specification of REV: rsgraph -> rsgraph, is given by:

REV(rep(nil)) = [⇓nil, ⇒nil, null, null]
∀n:nat,list.
REV(rep(n:l)) = [⇓(n:l), ⇒(n:l), hdrev(n:l), hd(n:l)]

Finally, we note that the distinction in the object specification between ϕ and τ is lost in the implementation; namely, rep(τ) = rsgraph and rep(ϕ) = rsgraph. The distinction could be kept by introducing further subsorts into ncol, ecol and sgraph; however, then we would be unable to specify *rep* and *abs* independently of the object specification. For example, in **Queue**, **Stack**, and several other specifications, the distinction between ϕ and τ coincides with the distinction between pairs of "empty" ncol and "empty" ecol elements (i.e. the classes containing constants $[]_n$ and $[]_e$) and pairs of "non-empty" ncol and ecol elements (i.e. the classes containing terms of form (x $+1_n$ N) and any term of sort ecol); whereas in **Binary_Tree**, the distinction between ϕ and τ coincides with the distinction between pairs of "non-empty" ncol and "empty" ecol elements (i.e. the classes containing terms of form (x $+1_n$ N) and the constant $[]_e$) and pairs of "non-empty" ncol and "non-empty" ecol elements (i.e. the classes containing terms of form (x $+1_n$ N) and (y $+1_e$ E)). Partially ordered sorts are introduced into the object specifications in order to cope with the partiality of selectors and to avoid explicit error-handling. Now that we have the situation where the specifications of the implemented operations are partial because of the representation invariant, partiality cannot be avoided and thus the distinction between the representation of ϕ and τ is no longer important.

5.10 Summary

In this chapter we have extended the storage relations of an ADT to *storage graphs*: directed graphs with a subset of nodes designated for efficient access. These nodes are referred to as *access nodes* and the operators which return their labels are *access operators*. Like storage relations, storage graphs can be specified equationally: the resulting specification is called **SGraph**. We specify **SGraph** so that we can implement

each object ADT by its corresponding **SGraph**: this means that we have a standard form of representation for each object ADT and a framework in which we can make further implementation decisions.

Two strategies for determining the access nodes and access operators are given. The first is a general, "automatic" strategy which does not incorporate any aspects of efficiency analysis, but it may be more appropriate for implicitly keyed ADTs. The second strategy involves an aspect of efficiency analysis by synthesising implementations of the object ADT by its corresponding **SRelation**. The form of the implementation has to be constrained in order to impose the structure of the object specification onto the **Nodes&Edges** part of **SRelation**. By treating the components of **Nodes&Edges**, the ADTs **Ecol** and **Ncol**, as keyed ADTs, we can derive exactly those positions which need to be accessed in the implementation. These positions are defined by given and auxiliary selectors: these are the access operators. This method of determining the access nodes is based on a naive trade-off of space for time and we give an optimisation which trades back some space for a constant increase in time.

Finally, we have described how implementation involve three kinds of program synthesis: synthesis using the equational theory, synthesis using the inductive theory, and synthesis using auxiliary operations. When the respective ADTs can be organised into canonical term rewriting systems, then the synthesis using the equational theory can be automated by the strategy described in [KaS 85]. When the synthesis uses the inductive theory, then we have described how the synthesis can be automated using the inductive inference algorithm from [Lan 87].

Chapter Six
Implementation by Imperative Language

In this chapter we consider the implementation of storage graphs in an imperative programming language. We discuss how to choose data structures and representation mappings for storage graphs which can lead to (time) efficient implementations of object specifications. An imperative programming language is specified as an ADT and we give the implementations of the storage graphs of two keyless specifications as examples. Finally, we discuss the circumstances under which implementations with constant time complexity can be synthesised.

6.1 Implementation in an Imperative Language

As we have discussed in chapter one, the synthesis of an imperative implementation for an ADT consists of two steps: i) the choice of data structures and the representation mapping, and ii) the definition of procedures and functions which implement the derived sort and primitive sort operators respectively.

As an informal example, when the object specification has derived sort operator `p: τ δ -> τ` and primitive sort operator `f: τ -> δ`, then we aim to synthesise Pascal-like type declarations, a procedure `P` and function `F`:

```
type    T = ...
        D = ...

procedure P(var x:T; y:D);
begin
    ...
end;

function F(x:T):D;
begin
    ...
end;
```

such that `P` and `F` are *correct* and *efficient*.

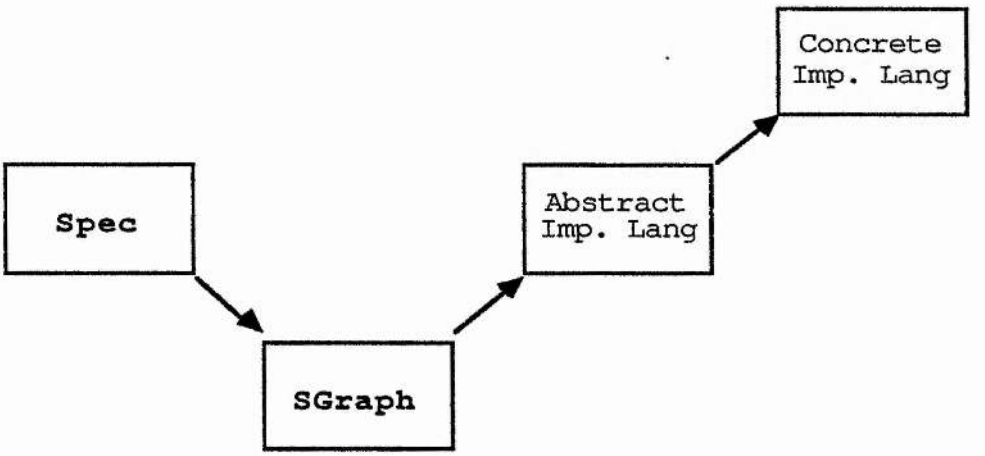
Namely, assuming that T represents τ, D represents δ, P represents p, and F represents f, then P and F have reasonable time complexity and for t:T and d:D, P fulfills the specification

{true} P(t,d) {abs(t')= p(abs(t),d)}

and F fulfills the specification

$(\forall t:T)$ abs(F(t)) = f(abs(t))

In order to discuss correctness, we need either an algebraic semantics for the imperative language, or (Hoare-like) proof rules; in order to discuss efficiency, we need the time and space requirements of the language constructs. For compatibility with our notion of correctness, we choose an algebraic semantics. So, we must synthesise the procedures and functions at a more abstract level: we need to view the imperative language as an ADT. Specifications of (some aspects of) imperative languages have been given, for example, in [BrW 80] and [GoP 81]; we have the additional difficulties of pointers and user-defined types. The specification of the latter is particularly complicated; moreover, we must choose the level of abstraction such that the choice of representing data structure can be discussed. Indeed, given the storage graph representation of an ADT, it would be easier to synthesise a represention in a low level assembly language or machine code than to synthesise a representation in a high level language such as Pascal. Our implementation process is a mixture of both 'compilation' and 'decompilation' as described by the following diagram where the 'compilation' process from **Spec** to **SGraph** has been described in the preceding chapters:



In our approach, we try to choose data structures which efficiently represent the sort *sgraph* and some of its operations. For a given object ADT, we choose a data structure and representation mapping which allows the storage relations, access operations, and some of the **Ncol** and **Ecol** operations, to be implemented by operations, or compositions thereof (without recursion), from the imperative language. The motivation for such a choice derives from the form of the implementations of the object operations in **Spec/SGraph**; the result is efficient implementations and a simple implementation synthesis.

For example, consider the **SGraph** implementation of an object operator op with arity τ -> τ by an operator OP: rsgraph -> rsgraph. Because of our constraints on the form of implementations, we can ensure that OP is specified by equations of the form:

$$OP(\langle \Downarrow t, \Rightarrow t, f_1(t), \ldots, f_n(t) \rangle) = \langle e_1, e_2, g_1, \ldots, g_n \rangle$$

where $e_1, e_2, g_1, \ldots, g_n$ are expressions which may depend on $\Downarrow t, \Rightarrow t, f_1(t), \ldots, f_n(t)$ (and any other access nodes if the access nodes have been optimised).

Our aim is to represent the components of sgraph in the imperative language such that representations of $f_1(t), \ldots, f_n(t)$ are accessible in constant time, and the representation of $\Rightarrow t$ has the property that when $x \Rightarrow_t y$ (for ok(t)) then the representation of x is accessible from the representation of y in constant time. If we can then synthesise representations of the operations which occur in $e_1, e_2, g_1, \ldots, g_n$ by primitive operations in the imperative language, or compositions thereof (without recursion), then we will have a constant time imperative implementation for OP. If an equation for OP is recursive, or if operations which occur in $e_1, e_2, g_1, \ldots, g_n$ have to be implemented by derived operators which are specified recursively, then we may not be able to synthesise a constant time imperative implementation. Moreover, when there is a straightforward representation mapping for each of $\langle \Downarrow t, \Rightarrow t \rangle, g_1, \ldots, g_n$, then most of the work of synthesising an implementation has already been done during the synthesis of **Spec/SGraph**.

As an example, consider the ADD operation from **Queue/SGraph**, (which we give now by one equation):

$\forall q$:queue, x_5:nat.

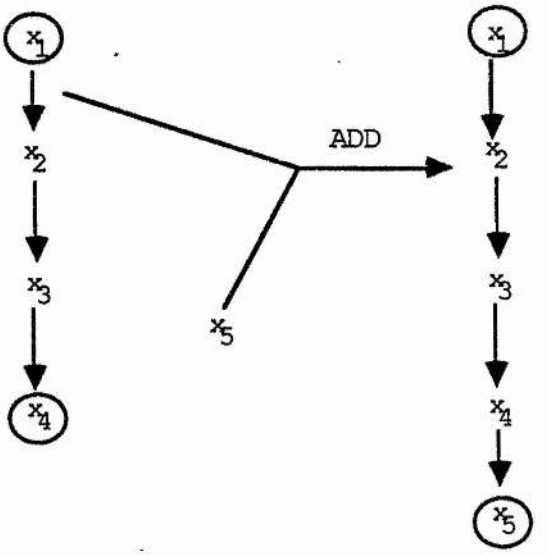
ADD(rep(q), x_5) =

if (q=eq)

then $\langle x_5 +1_n \Downarrow q, \Rightarrow q, x_5, x_5 \rangle$

else $\langle x_5 +1_n \Downarrow q, \langle last(q), x_5 \rangle +1_n \Downarrow q, front(q), x_5 \rangle$

When we represent ADD graphically as an operation on storage graphs (directed graphs with access nodes encircled), then we can easily see the relationship between the operands and their images under the operation, i.e.



We can see that $\Rightarrow_q$, x_5 and $\Rightarrow_{add(q,x5)}$ are informally related by:

$\Rightarrow_{add(q,x5)}$ is $\Rightarrow_q$ plus the node x_5 and an edge from last(q) to x_5, with the access node given by *last* updated to x_5.

Informally, it is quite easy to see how an imperative procedure could be derived from this relationship; i.e.

```
procedure ADD(var Q:T; x5:D);
begin
add node x5 to Q;
add edge from last(Q) to x5 to Q;
update last(Q) to x5;
end;
```

Clearly ADD (informally) satisfies the specification

$$\{\text{true}\}\ \texttt{ADD}(Q, x_5)\ \{\texttt{abs}(Q') = \texttt{add}(\texttt{abs}(Q), x_5)\}$$

In order to discuss the efficiency of this procedure, and to further develop the example, we must consider the choice of data type T.

Recall the data structure paradigm of entry points and implementation structures mentioned in §1.2.3 . If we choose a data structure with entry points which correspond to access operators, and an implementation structure with the similar properties as the storage type of the object ADT, (for example, a linear order), then there is a straightforward way in which imperative implementations can be derived from the equations specifying the **Spec/SGraph** operations. Namely, a change to the set of nodes is represented either by an allocation and initialisation of storage, or a deallocation of storage. A change to the set of edges is represented by updating the representation of the storage relation, and a change to an access node is represented by updating an entry point.

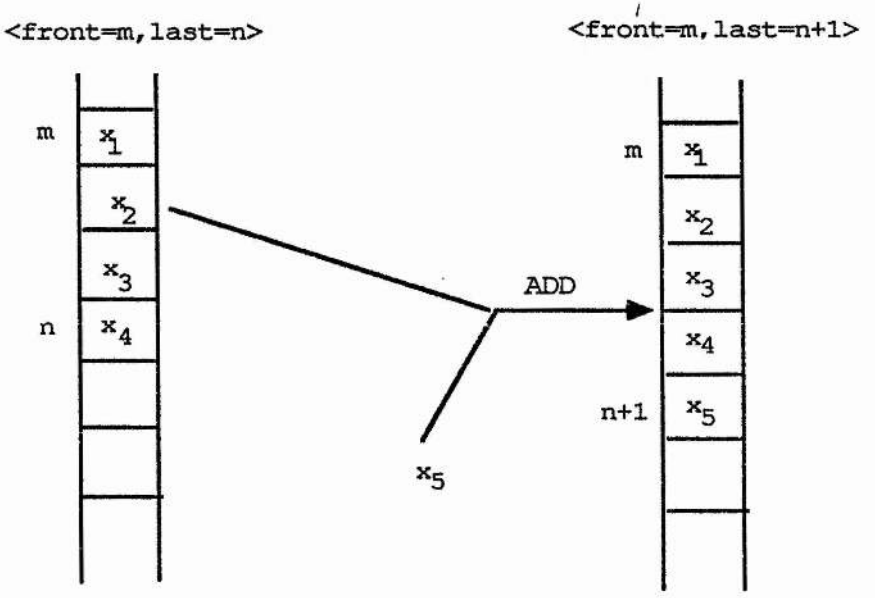
When we define the representation of queue by the data type of arrays with two indices; i.e.

```
type    Queue = record
        s: array[1..max] of integer;
        front,last : integer
        end;
```

then we can define the body of ADD by:

```
procedure ADD(var q:Queue;n:integer);
begin
with q do
     if last=max then last:=1 else last:=last+1;
     if front=last then Overflow else q[last]:=n
end;
```

which we represent pictorially as:



We note that in this representation, the storage relation $\Rightarrow_t$ is represented by adjacent positions in the array and so we do not have to make the addition of an edge explicit; i.e. (for `ok(t)`) if $x \Rightarrow_t y$ then (position of `x`) = (position of `y`) -1.

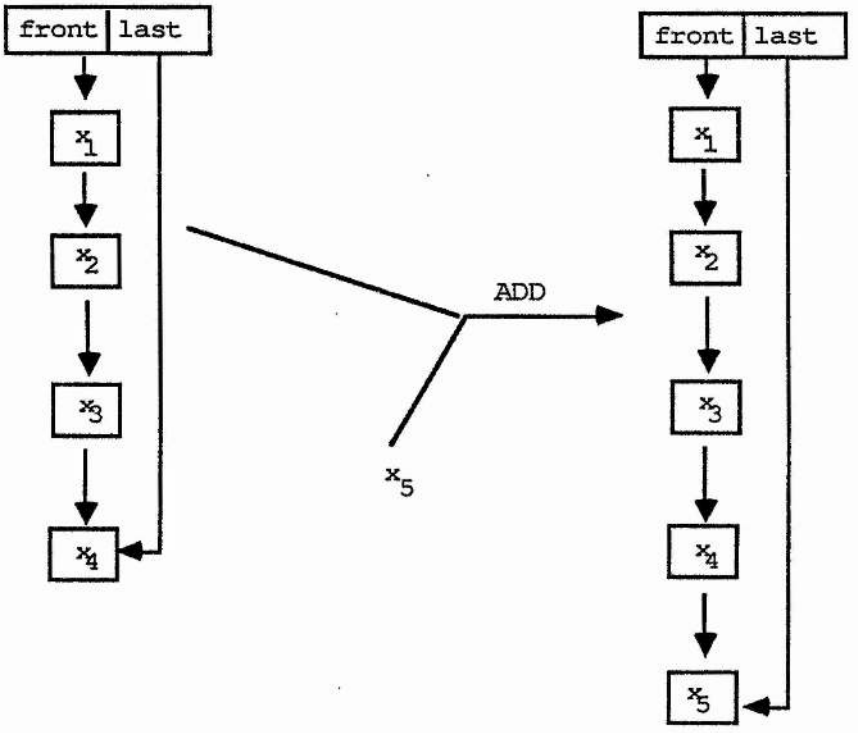
When we define the representation of `queue` by a linked data structure with two pointers into a linked structure; i.e.

```
type     List = record
         item:integer;
         next: ^List
         end;

type     Queue = record
         front, last : ^List
         end;
```

then ADD is given by:

```
procedure ADD(var q:Queue;n:integer);
var p:^List;
begin
new(p); p^.item:=n; p^.next:=nil;
with q do
    begin  if front=nil then front:=p else last^.next:=p;
    last:=p
    end
end
```

which we represent pictorially as:



We note that in this representation we have to make the addition of the new edge explicit because the storage relation is represented by pointers; i.e. (for `ok(t)`) if $x \Rightarrow_t y$, then (cell containing `x`)`^.next` = (cell containing `y`).

In the remainder of this chapter, we consider the formal derivation of imperative implementations; in the next section we consider the choice of data structures for keyless specifications with finite storage relations.

6.3 Choosing Data Structures for Keyless Specifications

In the absence of further information concerning the dynamic use of a given object ADT, we will choose the data structures in the imperative language according to the storage type of the ADT and the access operators in the storage graphs of the ADT. In this section, we suggest data structure choices under some circumstances; the circumstances which we consider are common, but not exhaustive.

We are primarily concerned with the selection of linked data structures. Moreover, we shall only consider the implementation of keyless ADTs with finite storage relations; implementations for implicitly keyed ADTs are not considered because the choice of data structures for efficient implementations of these ADTs must take into account further information concerning the ordering on the primitive sort. In the following subsections we describe a method of choosing linked data structures; this method will be used in our example implementations.

6.3.1 Linked Data Structures

We adopt the convention that a linked data structure consists of a ***head cell*** data structure and a ***data cell*** data structure. Data cells form the implementation structure whilst a head cell consists of a tuple of entry points, or pointers, into the implementation structure. A data cell type is a (Pascal-like) record with one "contents" field: a field whose type is the representation of δ, and one or more pointer fields (of type data cell). A head cell type is a (Pascal-like) record with one or more pointers (of type data cell).

Essentially, the number of pointers in the data cell type corresponds to the regularity of the object ADT; the number of pointers in the head cell correspond to the number of access operators. The intention is that the representation is defined such that the pointers in the head cell ensure that each access node can be reached in constant time, and the pointers in the data cell ensure that for a given derived element t and primitive element x, if y is related to x in the storage relation at t, then in the representation of t, the data cell containing y can be reached from the data cell containing x in constant time.

When the primitive part of the object specification is not anarchic, (i.e. there are primitive equations), then this method may result in an inefficient implementation (with respect to space). We may find a more efficient data structure by considering the storage type of a sub-specification.

6.3.1.1 Object Specifications with Anarchic Primitive Specifications

When the primitive specification of a given object specification is anarchic, and has n-regular or (m:n)-regular storage type, then we define the data cell to be a record consisting of one field of the representation of δ and n fields of type pointer to data cell. When the object specification has p (possibly optimised) access operators, then the head cell consists of p fields of type pointer to data cell.

For example, the **Queue** specification has 1-regular storage type and 2 access operators; thus we choose the linked data structure given in §6.2. As another example, the **Binary_Tree_2** example has 2-regular storage type and 1 (optimised) access operator; thus we choose the following linked data structure:

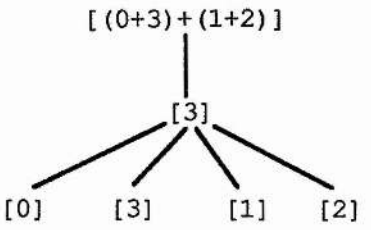
```
type    Datacell = record
        contents   : integer;
        daughter1  : ^Datacell;
        daughter2  : ^Datacell;
        end;

type    Headcell = record
        root : ^Datacell
        end;
```

6.3.1.2 Object Specifications with Primitive Equations

Recall that when considering the storage type of an ADT, we only consider generator terms with unique occurrences of primitive sorted subterms (c.f. §3.2.1). As we have seen in chapter 5, this restriction does not always ensure that there is an injection between the primitive sorted subterms of the derived terms and the primitive sorted subterms of the storage relations of those terms; i.e. the *ncol* and *ecol* terms. Namely, when the primitive specification has some structure, then this may be used to specify selection; i.e. the (classes in the) contents set of t may not necessarily contain subterms of t. For example, in the **Binary_Tree_1** specification, `root(t)`, when `t` has normal form `comb(comb(n(0),n(3)),comb(n(1),n(2)))`, is congruent to `(0+3)+(1+2)`. Moreover, the congruence generated by **ANat** forces `root(left(t))` and `root(right(t))` to be identified when they might not be identified given another term `t`. Thus `[root(left(t))]` is related to four classes in the storage relation at `t`:

`[root(right(root(left(t))))]`,`[root(left(root(left(t))))]`,
`[root(right(root(right(t))))]`, and `[root(left(root(right(t))))]`.

Namely, we have the following situation:



Given another term `t`, we might find that `[root(t)]` is only related to two other classes, or indeed, to more than four other classes.

Clearly, although it would be correct to store the representations of `[root(left(t))]` and `[root(right(t))]` in the same data cell; it would not be efficient to do so unless for all `t`, `root(left(t))` and `root(right(t))` are congruent. It would not be efficient because if for all `t`, `root(left(t))` and `root(right(t))` are not congruent, then (the implementation of) every tree manipulation operation (i.e. combination, left and right subtree operations) must depend on the structure and nature of its operands and therefore consist of several "branches".

We suggest that a more efficient data structure may be found by "forgetting" the structure on the primitive sort and considering the regularity of the storage type of the subspecification. We "forget" the structure specified by the primitive specification by eliminating the primitive equations.

Thus, for a given a specification (Σ,E) over primitive specification (Σ_p,E_p), when the storage type of $(\Sigma, E\backslash E_p)$, where \ is set difference, is n-regular or (m:n)-regular, then we define the data cell to be a record consisting of one field of the representation of δ, and n fields of type pointer to data cell. The head cell is chosen as described for the anarchic case. (We note that when (Σ,E) is anarchic, then $E = E\backslash E_p$).

To conclude this section on linked data structures, we note that when the storage type is (m:n)-regular and the difference between m and n is large, then it may be more (space) efficient to dynamically allocate space for the links in the datacell; i.e. to give a list of pointers instead of a fixed number (n) of pointers.

6.3.2 Array Based Data Structures

In this section we briefly consider the choice of array-based data structures.

When choosing these kinds of data structures, we need to decide on the base type and the method of space allocation within the array.

If we can define a relation on the index type and an abstraction/representation relationship between that relation and each (restricted) storage relation, then we can define the base type as the representation of δ. When this is possible, then the storage relation can be represented by the relation on the index type; moreover, this relationship will be reflected in the method of space allocation. Clearly, such a relation on the index type will have many, if not all, of the properties in the storage type of the object specification.

For example, when a given object ADT has singly-linked linear storage type and the index type is the integers, then we may define a singly-linked linear relation *is_1_less_than*, where x *is_1_less_than* y iff x = y-1, and a representation mapping (informally) given by:

when A is an array and `t` is a derived sort term s.t. `ok(t)`,

if for all primitive sort terms `x`, `y`, such that $[x] \Rightarrow_t [y]$,

(A[i]=x and A[j]=y) implies (i *is_1_less_than* j),

then A represents `t`.

Of course, in order to formalise the representation/abstraction relationship, we must specify exactly which positions in the array are being used to represent `t`; we do not care about the relationship between positions which are not being used in the representation.

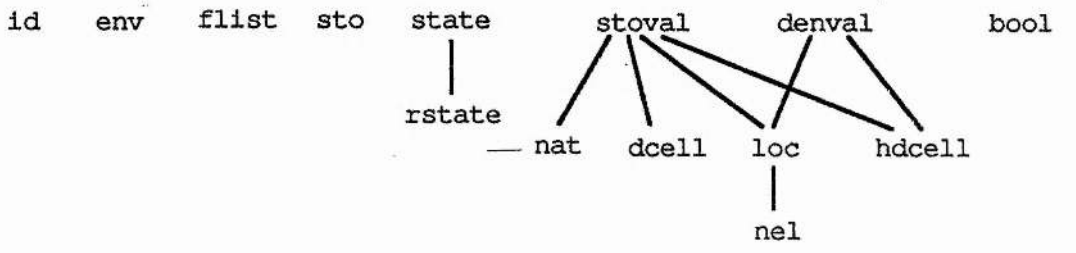
If such a relation on the index type cannot be found, then the representation of the storage relation must be specified more explicitly. For example, if the object specification is (m:n)-regular, then we can define the base type to be a record consisting of one "contents" field and n fields of the index type. We note that when we use such a data structure, then the allocation of space within an array does not depend on any property of the storage type.

Obviously, there is much scope for defining different kinds of representations between storage relations and array-based data structures: good representations will exploit various properties of the storage type of the object ADT.

6.4 Specifying an Imperative Language with Linked Data Structures

In this section we specify some features of an imperative programming language with linked data structures. The language specified below is the kernel implementation language for the examples considered in the following section; for each example, we will enrich the specification with the appropriate head cell data type and data cell data type. For simplicity, the language includes only the essential features: identifiers, environments, stores and free location lists; locations are not typed and we have not specified any error-handling. Thus, the specification is not sufficiently-complete w.r.t. any observable sorts such as **Nat** and **Bool**.

The denotable values of the language include locations and head cells; the storable values include the natural numbers, locations, head cells and data cells. The former are given by the sort *denval* and the latter by the sort *stoval*; we give the sorts of the language below:



The language is given by the ADT **Program_State** which is built up from the specifications **Ide**, **Values**, **Stores&Env**, and **Flist**.

```
spec Ide =

data sorts id

opns  T : id                                  /*hdcell id*/
      p : id                                  /*dcell id*/
      q : id
      + : id -> id                            /*infinite ids*/
      n : id
end
```

```
spec Values = enrich Nat + Bool + Ide by

data sorts   denval, stoval, loc, nel, hdcell, dcell

subsorts     hdcell ≤ denval, loc ≤ denval, nat ≤ stoval,
             dcell ≤ stoval, hdcell ≤ stoval, loc ≤ stoval

opns  lo  : nel                                    /*location opns*/
      +   : nel -> nel
      nil : loc                                    /*nil pointer*/

end


spec Stores&Env = enrich Values by

data sorts   sto, env

opns                                               /*store opns*/
      esto     : sto                               /*empty store*/
      initsto  : sto                               /*initial store*/
      _:=_,_   : loc stoval sto -> sto             /*assignment*/
      val_in_  : loc sto -> stoval                 /*stored value*/

                                                   /*env opns*/
      eenv          : env                          /*empty env*/
      initenv       : env                          /*initial env*/
      _bindto_ in_  : id denval env -> env         /*add id binding*/
      lookup        : id env -> denval             /*find binding*/

eqns
                                                   /*store eqns*/
∀l1,l2:nel,v:stoval,s:sto.
val l1 in (l2:=v,s)  = v                           if (l1=l2)

∀l1,l2:nel,v:stoval,s:sto.
val l1 in (l2:=v,s)  = val l1 in s                 if ~(l1=l2)


                                                   /*env eqns*/
∀i1,i2:nel,v:denval,e:env.
lookup(i1,(i2 bindto v in e)) = v                  if (i1=i2)

∀i1,i2:nel,v:denval,e:env.
lookup(i1,(i2 bindto v in e)) = lookup(i1,e)       if ~(i1=i2)

end


spec Flist = enrich Stores&Env by

data sorts   flist

opns  elist    : flist                             /*empty list*/
      initlist : flist                             /*initial flist*/
      _::_     : nel flist -> flist                /*add to flist*/
      -        : flist -> flist                    /*remove locn*/
      new      : flist -> nel                      /*new locn*/
```

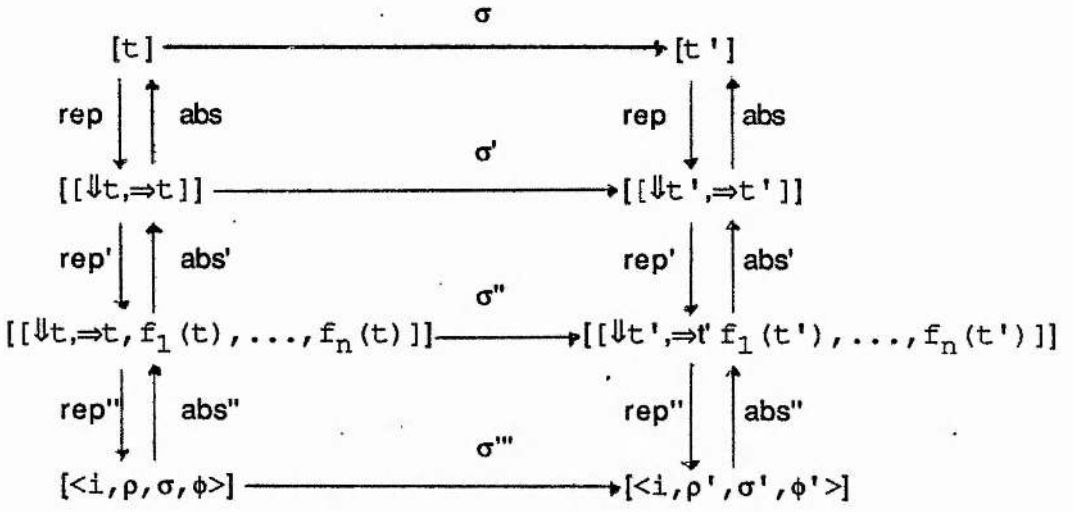
```
eqns
∀l:nel,f:flist.
new(l::f) = l

∀l:nel,f:flist.
-(l::f) = f

end
```

```
spec Program_State = enrich Flist by

data sorts   state, rstate

subsorts     rstate ≤ state

opns
     <_,_,_,_>   : id env flist sto -> state   /*make state*/
     initstate   : state                       /*initial state*/
     new_in_     : id state -> state           /*new id decl*/
     i           : state -> id                 /*selectors*/
     e           : state -> env
     f           : state -> flist
     s           : state -> sto

eqns
∀l:i1:id,e1:env,f1:flist,s1:sto.
i<i1,e1,f1,s1> = i1
∀l:i1:id,e1:env,f1:flist,s1:sto.
e<i1,e1,f1,s1> = e1
∀l:i1:id,e1:env,f1:flist,s1:sto.
f<i1,e1,f1,s1> = f1
∀l:i1:id,e1:env,f1:flist,s1:sto.
s<i1,e1,f1,s1> = s1

∀l:i1,s1:state.
new i1 in s1 =
     <i(s1), i1 bindto (next(f(s1))) in e(s1), s(s1), -(f(s1))>

end
```

The operations specified above correspond to the usual, simple constructs of an imperative programming language. We note that iterative constructs have not been included (although they may of course be specified as derived operators). Therefore, we may assume that each operation takes a constant amount of time to execute; i.e. $O(1)$.

6.5 Example Implementations

Finally, we consider the synthesis of an implementation of an object ADT by the imperative language. Namely, when **ImpLang** is the specification of the imperative language (**Program_State** enriched with suitable data structures), we synthesise **Spec/ImpLang**. So, we end up with each τ-sorted element of **Spec** being represented by a program state (a tuple consisting of an identifier, environment, store and freelist); derived sort operations are represented by procedures (operations on program states). To summarise, when $\sigma : \tau \rightarrow \tau$ and $\sigma[t]=[t']$, (σ generalises to any arity), then we have the following (commuting) diagram:



The specification of the representation and abstraction mappings between the first three layers, i.e. between **Spec**, **SRelation**, and **SGraph**, has been quite straightforward; the specifications of rep" and abs" are considerably more complex. We do not attempt to specify these mappings in general, as we did for abs, rep, abs', and rep', because they depend on the chosen data structures in the imperative language. Moreover, we note that for most object specifications, the abstraction mapping (from imperative language to storage graphs) is easier to specify than the representation mapping.

In our examples, we specify the usual relationship between the representation and abstraction mappings (which we just call *rep* and *abs* resp.) and we give some equations for the abstraction mapping. We also include an axiom to identify program states which abstract to the same storage graph.

In addition, for readability, we adopt the convention that *initenv* and *initsto*, (the "initial" environment and "initial" store resp.), are used in the equations which

specify the (representations of the) τ-sorted constants.

The specification of **ImpLang** (**Program_State** enriched with the chosen (abstract) data structures) is given first. The specification of **Spec/ImpLang** follows. For clarity, each (new) operation in the latter specification is given in concrete syntactic form. In the following section, (§6.6), we discuss the efficiency of the examples and describe the circumstances under which constant time implementations can be synthesised.

6.5.1 Queue Implementation

Recall that the chosen data structure for **Queue** is the linked data structure with two pointers in the head cell and one pointer in the datacell. The specification of **ImpLang** using this data structure, (including the relevant abstraction/representation mappings), is given by:

spec **Imp_Lang = *enrich*** **Program_State + Queue/SGraph *by***

```
opns                                                    /*head cell*/
      mkhcell      : loc loc -> hdcell
      _.front      : hdcell -> loc
      _.last       : hdcell -> loc
                                                        /*data cell*
      mkdcell      : nat loc -> dcell
      _.next       : dcell -> loc
      _.contents   : dcell -> nat


      rep          : rsgraph -> rstate                  /*representation*/
      abs          : rstate -> rsgraph                  /*abstraction*/

      absf         : loc -> nat                         /*aux. abs opns*/
      absl         : loc -> nat
      nodes        : state -> ncol
      edges        : state -> ecol
      nextn        : loc sto -> ncol
      nexte        : loc sto -> ecol

eqns
                                                        /*data structs*/
∀l1,l2:loc.
(mkhcell(l1,l2)).front =l1
∀l1,l2:loc.
(mkhcell(l1,l2)).last = l2

∀l1:loc,n:nat.
(mkdcell(n,l1)).contents = n
```

```
∀l1:loc,n:nat.
(mkdcell(n,l1)).next = l1
                                                    /*abstraction*/

∀g:rsgraph.
abs(rep(g)) = g

∀s1,s2:state.
(abs(s1)=abs(s2)) => (s1=s2)

∀s1:rstate.
abs(s1) = [nodes(s1),edges(s1),
              absf(val lookup(i(s1),e(s1)) in s(s1)),
              absl(val lookup(i(s1),e(s1)) in s(s1))]


absf(nil) = null
∀l:nel.
absf(l) = l.front

absl(nil) = null
∀l:nel.
absl(l) = l.last

∀i:id,e1:env,s1:sto,f1:flist
nodes(<i1,e1,f1,s1>) = 1
      (val(val lookup(i,e) in s).front) in s1).contents +l_n
          nextn((val(val lookup(i1,e1) in s1).front)
                  in s1).next),s)

∀i:id,e1:env,s1:sto,f1:flist
edges(<i1,e1,f1,s1>) = nexte((val lookup(i,e) in s1).front,s1)

∀s:sto
nextn(nil,s) = []_n
∀l:nel,s:sto
nextn(l,s) = (val l in s).contents +l_n
                  nextn((val l in s).next),s)

∀s:sto
nexte(nil,s) = []_e
∀l:nel,s:sto
nexte(l,s) =
      <(val l in s).contents, val((val l in s).next).contents>
          +l_e nexte((val l in s).next).s)
                                  if ~(((val l in s).next)=nil)

∀l:nel,s:sto
nexte(l,s) = []_e                 if (((val l in s).next)=nil)


end
```

Now, we consider the specification of the operations in **Queue/ImpLang**. It is quite apparent that the equational framework is inappropriate for presenting (what amounts to) the denotational description of operations (procedures) in an imperative language; a meta language construct for local bindings simplifies the presentation somewhat. For example, we can synthesise the following (conditional) equation for the ADD operation, where ADD is the (overloaded) implementation of add; i.e. ADD : rstate nat -> rstate.

```
∀s1:rstate,n:nat.
ADD(s1,n) =
<i(s1), e(s1),-(f(s1)),
     val(lookup(i(s1),e(s1)) in s(s1)).last
        := lookup(p,p bindto new(f(s1) in e(s1)) in
           val(val(lookup(i(s1),e(s1)) in s(s1)).last).next
              := lookup(p,p bindto new(f(s1)) in e(s1)) in
                 lookup(p,p bindto new(f(s1)) in e(s1))
                              := mkcell(n,nil) in s(s1)>

           if ~((val lookup(i(s1),e(s1)) in s(s1)).front = nil)
```

The introduction of local (meta) variables and their bindings allows for a much more readable presentation; i.e.

```
∀s1:rstate,n:nat.
ADD(s1,n) =
<i(s1), e(s1),-(f(s1)),
     T'.last := lookup(p,e') in
        (val(T'.last) in s(s1)).next := lookup(p,e')in
           lookup(p,e'):= mkcell(n,nil) in s(s1)>

                                        if ~(T'.front = nil)
```

where

$T' =_{def}$ val(lookup(i(s1),e(s1)) in s(s1)
$e' =_{def}$ p bindto new(f(s1)) in e(s1)

In the following specifications, we use the ***where*** notation. The scope of a meta-variable binding is the equation preceding the ***where*** clause; when $e_1,...,e_n$ are equations, we use the abbreviation

whereblock $e_1,...,e_n$ ***where*** $v =_{def}$ exp

to stand for

e_1 ***where*** $v =_{def}$ exp, e_2 ***where*** $v =_{def}$ exp, etc.

```
spec Queue/Imp_Lang = enrich Imp_Lang by

opns      EQ      : rstate
          ADD     : rstate nat -> rstate
          DEQUEUE : rstate -> rstate
          FRONT   : rstate -> nat

eqns

initenv = T bindto l0 in eenv

initsto = l0 := mkhdcell(nil,nil) in esto

EQ = <T,initenv,initflist,initsto>

whereblock

∀s1:rstate,n:nat.
ADD(s1,n) =
<i(s1), e(s1),-(f(s1)),
     T'.last := lookup(p,e') in
          (val(T'.last) in s(s1)).next := lookup(p,e')in
               lookup(p,e'):= mkcell(n,nil) in s(s1)>

                                        if ~(T'.front = nil)

∀s1:rstate,n:nat.
ADD(s1,n) =
<i(s1), e(s1),-(f(s1)),
     T'.last := lookup(p,e') in
          T'.front := lookup(p,e') in s(s1)>
               lookup(p,e'):= mkcell(n,nil) in s(s1)>

                                        if (T'.front = nil)


∀s1:rstate.
FRONT(s1) =
(val (T'.front) in s(s1)).contents

∀s1:rstate.
DEQUEUE(s1) =
<i(s1),e(s1),(T'.front)::f(s1),
     T'.front := next in (T'.last := nil in s(s1))>

                                   if (T'.front=nil) ∧ (next =nil))


∀s1:rstate.
DEQUEUE(s1) =
<i(s1),e(s1),(T'.front)::f(s1),
     T'.front := next in  s(s1)>

                                   if (T'.front=nil) ∧ ~(next=nil))
```

where

```
T'      =def val(lookup(i(s1),e(s1)) in s(s1)
e'      =def p bindto new(f(s1) in e(s1)
next =def val(val T'.front in s(s1)) in s(s1)
```

To conclude this example, we give the operations specified above in a concrete syntactic form:

```
var T:Queue;

procedure EQ;
begin T.front := nil; T.last := nil
end;

procedure ADD(var T:Queue; n:Integer);
var p:^List;
begin
new(p); p^.contents := n; p^.next := nil;
if T.front = nil then T.front := p else (T.last)^.next := p;
T.last := p
end;

procedure DEQUEUE(var T:Queue);
var p:^List;
begin
if T.front <> nil
     then
     begin
     p := T.front; T.front := (T.front)^.next;
     if T.front = nil then T.last := nil;
     dispose(p)
end;

function FRONT(T:Queue);
begin
FRONT := (T.front)^.contents
end;
```

6.5.2 Reversible List Implementation

Recall that the **Reversible_List_1** specification has (1:2)-regular storage type (cf. §3.3) and two access operators: *hd* and *hdrev* (cf. §5.6.4.). Using the method described in §6.3, we choose the following (concrete) data structure:

```
type Datacell =  record
                 contents : integer;
                 next1 : ^Datacell;
                 next2 : ^Datacell
                 end;

     Headcell =  record
                 front : ^Datacell;
                 last : ^Datacell
                 end;
```

The specification of **ImpLang** using this data structure, (including the relevant abstraction/representation mappings), is given by:

```
spec Imp_Lang = enrich
        Program_State + Reversible_List_1/SGraph by

opns                                                    /*head cell*/
     mkhcell      : loc loc -> hdcell
     _.front      : hdcell -> loc
     _.last       : hdcell -> loc
                                                        /*data cell*
     mkdcell      : nat loc -> dcell
     _.next1      : dcell -> loc
     _.next2      : dcell -> loc
     _.contents   : dcell -> nat


     rep          : rsgraph -> rstate                   /*representation*/
     abs          : rstate -> rsgraph                   /*abstraction*/

     absf         : loc -> nat                          /*aux. abs opns*/
     absl         : loc -> nat
     nodes        : state -> ncol
     edges        : state -> ecol
     nextn        : loc sto -> ncol
     nexte        : loc sto -> ecol

eqns
                                                        /*data structs*/
∀l1,l2:loc.
(mkhcell(l1,l2)).front =l1
∀l1,l2:loc.
(mkhcell(l1,l2)).last = l2

∀l1,l2:loc,n:nat.
(mkdcell(n,l1,l2)).contents = n

∀l1,l2:loc,n:nat.
(mkdcell(n,l1,l2)).next1 = l1

∀l1,l2:loc,n:nat
(mkdcell(n,l1,l2)).next2 = l2
```

```
                                                        /*abstraction*/
∀g:rsgraph.
abs(rep(g)) = g

∀s1,s2:state.
(abs(s1)=abs(s2)) => (s1=s2)

∀s1:rstate.
abs(s1) = [nodes(s1),edges(s1),
              absf(val lookup(i(s1),e(s1)) in s(s1)),
              absl(val lookup(i(s1),e(s1)) in s(s1))]


absf(nil) = null
∀l:nel.
absf(l) = l.front

absl(nil) = null
∀l:nel.
absl(l) = l.last
```

whereblock

```
∀i:id,e1:env,s1:sto,f1:flist
nodes(<i1,e1,f1,s1>) =
      (val(val lookup(i,e) in s).front) in s1).contents +1n
          nextn((val(val lookup(i1,e1) in s1).front)
               in s1).next2),s)
      plusn
      (val(val lookup(i,e) in s).lastt) in s1).contents +1n
          nextn((val(val lookup(i1,e1) in s1).last)
               in s1).next1),s)
                                        if (first.next1=nil)


∀i:id,e1:env,s1:sto,f1:flist
nodes(<i1,e1,f1,s1>) =
      (val(val lookup(i,e) in s).front) in s1).contents +1n
          nextn((val(val lookup(i1,e1) in s1).front)
               in s1).next1),s)
      plusn
      (val(val lookup(i,e) in s).last) in s1).contents +1n
          nextn((val(val lookup(i1,e1) in s1).last)
               in s1).next2),s)
                                        if (first.next2=nil)



∀i:id,e1:env,s1:sto,f1:flist
edges(<i1,e1,f1,s1>) =   nexte((val lookup(i,e) in s1).front,s1)
                         plusn
                         nexte((val lookup(i,e) in s1).last,s1)

∀s:sto
nextn(nil,s) = []n
```

```
∀l:nel,s:sto
nextn(l,s) = (val l in s).contents +l_n
                  nextn((val l in s).next2),s)

                                        if (first.next1=nil)

∀l:nel,s:sto
nextn(l,s) = (val l in s).contents +l_n
                  nextn((val l in s).next1),s)

                                        if (first.next2=nil)

∀s:sto
nexte(nil,s) = []_e
∀l:nel,s:sto
nexte(l,s) =
      <(val l in s).contents, val((val l in s).next2).contents>
          +l_e nexte((val l in s).next2).s)

                                        if (first.next1=nil)

∀l:nel,s:sto
nexte(l,s) =
      <(val l in s).contents, val((val l in s).next1).contents>
          +l_e nexte((val l in s).next1).s)

                                        if (first.next2=nil)

∀l:nel,s:sto
nexte(l,s) = []_e                  if (((val l in s).next)=nil)
```

where

first $=_{def}$ val(val(lookup(i1,e1)) in s1).front in s1

end

Remark: The expressions (first.next1=nil) and (first.next2=nil) are tests for determining the interpretation of the pointer fields in the data cells .

The specification of **Reversible_List_1/ImpLang** follows; we note that the operation INSERT implements the operator _:_.

spec **Reversible_List_1/Imp_Lang** = ***enrich*** **Imp_Lang** ***by***

```
opns      NIL      : rstate
          INSERT   : rstate nat -> rstate
          REV      : rstate -> rstate
          TL       : rstate -> rstate
          HD       : rstate -> nat
```

eqns

```
initenv = T bindto 10 in eenv

initsto = 10 := mkhdcell(nil,nil) in esto

NIL = <T,initenv,initflist,initsto>
```

whereblock

```
∀s1:rstate,n:nat.
INSERT(s1,n) =
<i(s1), e(s1),-(f(s1)),
     T'.front := lookup(p,e') in
          T'last := lookup(p,e') in
               lookup(p,e'):= mkcell(n,nil,nil) in s(s1)>

                                        if (T'.front = nil)

∀s1:rstate,n:nat.
INSERT(s1,n) =
<i(s1), e(s1),-(f(s1)),
     T'.front := lookup(p,e') in
          (val(lookup(p,e')) in s(s1)).next2) := T'.front in
               first.next1 := lookup(p,e') in
                    lookup(p,e'):= mkcell(n,nil,nil) in s(s1)>

                                        if (first.next1 = nil)

∀s1:rstate,n:nat.
INSERT(s1,n) =
<i(s1), e(s1),-(f(s1)),
     T'.front := lookup(p,e') in
          (val(lookup(p,e')) in s(s1)).next1) := T'.front in
               first.next2 := lookup(p,e') in
                    lookup(p,e'):= mkcell(n,nil,nil) in s(s1)>

                                        if (first.next2 = nil)

∀s1:rstate.
FRONT(s1) =
first.contents

∀s1:rstate.
REV(s1) =
<i(s1),e(s1),f(s1),
     T'.front := T'.last
          T'.last := T'.front>

∀s1:rstate.
TL(s1) = s1                             if (T'.front=nil)

∀s1:rstate.
TL(s1) =
<i(s1),e(s1),(T'.front)::f(s1),
     T'.front := nil in (T'.last := nil in s(s1))>

                        if (~(T'.front=nil) ∧ (T'.front=T'.last))
```

```
∀s1:rstate.
TL(s1) =
<i(s1),e(s1),(T'.front)::f(s1),
     first.next2 := nil in
          (valfirst.next2 in s(s1)).next1
               := nil in s(s1)>

                                   if (first.next1=nil)

∀s1:rstate.
TL(s1) =
<i(s1),e(s1),(T'.front)::f(s1),
     first.next1 := nil in
          (valfirst.next1 in s(s1)).next2
               := nil in s(s1)>


                                   if (first.next2=nil)
```

where

```
T'        =def val(lookup(i(s1),e(s1)) in s(s1)
e'        =def p bindto new(f(s1) in e(s1)
firstt    =def val T'.front in s(s1)
```

To conclude this example, we give the operations specified above in a concrete syntactic form:

```
var T:Headcell;

procedure NIL;
begin T.front := nil; T.last := nil;
end;

procedure INSERT(var T:Headcell; n:Integer);
var p:^Datacell;
begin
new(p); p^.contents :=n; p^.next1 := nil; p^.next2 := nil;
if T.front = nil                          /*empty list*/
     then
          begin T.front := p; T.last := p; end
     else
          begin
          if (T.front)^.next1 = nil  /*find direction*/
               then
                    begin
                    (T.front)^.next1 := p; p^.next2 := T.front
                    end
               else
                    begin
                    (T.front)^.next2 := p; p^.next1 := T.front
                    end
          end;
T.front := p
end;
```

```
procedure TL(var T:Headcell);
var temp:^Datacell;
begin
if (T.front<>nil)                    /*empty list*/
     then
          begin
          temp := T.front;
          if T.front = T.last          /*1 elem list*/
               then
                    begin T.front := nil; T.last := nil end
               else
                    begin
                    with (T.front)^ do
                    if next1 = nil
                         then
                              begin
                              next2^.next1 := nil; next2 := nil
                              end
                         else
                              begin
                              next1^.next2 := nil; next1 := nil
                              end
                    end;
          dispose(temp)
          end
end;

procedure REV(var T:Headcell);
var temp:^Datacell;
begin
temp := T.last; T.last := T.front; T.front := temp
end;

function HD(T:Headcell):Integer;
begin
HD := (T.front)^.contents
end;
```

Remark: The expression `(T.front)^.next1 = nil` is evaluated often. In effect, this is a test to establish which field: `next1` or `next2`, currently represents the "direction" of the list; i.e. when we traverse the list starting from one end, we follow either the `next1` links or the `next2` links. The choice of links to follow depends on which end we start at. A further space-time optimisation would be to allocate a variable for storing the value of the expression `(T.front)^.next1 = nil`. The variable would only be updated (to another proper Boolean value) in the `REV` procedure because the other operations either preserve the value of the expression, or cause it to become undefined (i.e. `TL` might set `T.front` to `nil`).

The representation mappings between the storage graphs and the imperative languages are complex because of the nature of the imperative language. In general, we consider the representation of a storage graph as a composition of the representation of its components; i.e. we have three kinds of representation mappings: `repn:ncol -> state`, (the representation of nodes), `repe: ecol -> state` (the representation of edges), and `repa: ecol -> state` (the representation of access nodes). The operations on each of these components; i.e. operations on the nodes, edges, and access nodes, can affect the program state. For example, in the **Queue** example, we specify the representation of the $+1_n$ by:

```
∀n:nat,N:ncol.
repn(n +1_n N) =
<i(repn(N)),e(repn(N)),-f(repn(N)),
      lookup(p, p bindto new(f(repn(N))) := mkdcell(n,nil) in
           s(repn(N))>
```

. For a given **ImpLang**, (a suitable enrichment of **Program_State** by the chosen data structures) it is easy to see that the operations from **Nodes&Edges'** (cf. §5.6.2.1), *excepting* the operations *mapt* and *tgtn*, can be represented by compositions (without recursion) of operations from **ImpLang**. *mapt* and *tgtn* cannot be represented, (without recursion,) because we have no primitive operations for selecting the incoming pointers to a datacell; we can only select outgoing pointers.

We conclude that if for a given object **Spec**, we can give a presentation of **Spec/SGraph** such that it contains only operations from

- the object specification,
- the access operations,
- **Nodes&Edges'**, excluding *mapt* and *tgtn*,
- the operations hd_n, tl_n, hd_e, tl_e (cf. §5.8),

and

- we use the linked data structure selection method given in §6.3,

then it is possible to synthesise a constant time **ImpLang** implementation.

Often, we may be able to synthesise constant time implementations for specifications which do not meet these requirements. Our point is that in general, we can only predict efficient implementations when the derived operators are specified in a particular way.

In the **Queue** example, the specification of **Queue/SGraph** meets the above conditions and the example imperative implementation runs in constant time.

In the **Reversible_List_1** example, we find that the operation *tgtn* occurs in part of the **Reversible_List_1/SRelation5** specification which is included in the **SGraph** implementation. Specifically, *tgtn* occurs in an equation specifying the TL operation (cf. Example 5.11, §5.6.4). However, it is possible to give an alternate specification which does not include *tgtn*; namely:

```
∀n:nat,l:nel.
TL(rep(n:l)) =
      [⇓(n:l) \n hd(n:l),
      (⇒(n:l) \\e scen(⇒(n:l),hdn(h:l))) \e <hd(n:l),hdtl(n:l)>]
```

where hdtl is a new access operator:

```
∀n:nat,l:nel.
hdtl(n:l) = hd(l).
```

We justify our chosen data structures by considering only the 1-optimised access operators. hdtl is not a 1-optimised access operator and is specified (in the language of storage relations) by:

```
∀n:nat,l:nel.
hdtl(n:l) = adj(⇒(n:l),hd(n:l)).
```

This presentation meets our conditions and we see that our example implementation runs in constant time.

The remaining example specifications from Appendix Two conform to the conditions described above; excepting the **Binary_Tree** examples and **Circular_List_left**. The former specifications do not meet the conditions because the auxiliary "$child_n$" and "$child_e$" operations are required. The latter specification does not meet the conditions because *tgtn* occurs in the specification of hdshift(shift(n:l)), the specification of the effect of the *shift* operation on the *hdshift* access operator, (cf. Example 5.18, §5.9.1). If we do not insist that unused storage is explicitly released, then constant time implementations for the **Binary_Tree** examples can be synthesised. However, we cannot give a specification for hdshift(shift(n:l)) which conforms to the above conditions. At best, using our method, we can only synthesise a linear time implementation for **Circular_List_Left**. If efficiency is crucial, then it appears that an

implementation strategy based on the storage relations is not appropriate for this specification.

6.7 Summary

In this chapter we have considered the specification of an imperative language and the choice of data structures in that language. We have described a method for choosing linked data structures; the method depends on the properties of **SGraph** and the storage type of the object specification. The specification of the language with the chosen (abstract) data structures is called **ImpLang**.

Two example implementations are presented in detail: **Queue** and **Reversible_List_1**.

We have described the circumstances under which constant time implementations for object specifications can be synthesised using our method. The method leads to efficient implementations for most of our example keyless specifications. For example, the implementations for **Queue** and **Reversible_List_1** run in constant time, but the method does not lead to a constant time implementation for the **Circular_List_left** specification.

Chapter Seven
Conclusions and Future Work

7.1 Summary

In this thesis we have considered some of the problems of implementing keyless and implicitly keyed ADTs in an imperative programming language.

In the introduction we identified three aspects of the problem of implementing an ADT: the form of the desired implementation, the choice of implementing data structures (and the representation map between the ADT and the data structures), and the construction of the implementation. Early on, we decided to implement the primitive sort operations by pure functions and the derived sort operations by imperative procedures. So, we have concentrated on the second and third aspects: choosing data structures and constructing efficient and correct imperative procedures which use those data structures.

Our methodology for choosing the implementing data structures is based on the analysis of the algebraic semantics of the ADT. It involves partitioning the operators of the signature and defining a family of binary relations on the model of the ADT. The relations depend only on the partitioning; they are called *storage relations* and reflect an "observational" view of the ADT, with respect to the primitive sort. The storage relations serve as an intermediate representation of the object ADT; thus, when we define these storage relations equationally, we can consider every ADT as a specification of binary relations, or directed graphs.

In order to choose the implementing data structures, we first consider the storage relations as representations of the elements of the ADT; then, we determine which of the elements in the domains of the relations should be immediately accessible in order for the storage relations to be correct and efficient representations. These elements are called *access nodes* and they are defined by *access operators*. Two different strategies for determining them are described. The first strategy is a general one and does not depend on the given ADT; however, the nodes thus determined are only those required for a correct implementation. The second strategy may require human intervention because it involves constructing and analysing the (partial) implementation of an object specification by its storage relations. The advantage of this method over the first is that both efficiency and

correctness determine the selection of the access nodes. A storage relation and its access nodes comprise a *storage graph*. The choice of data structures and representation map depends on properties of the storage relations and storage graphs.

We have described one particular implementation strategy. In this strategy, the chosen data structure is a product, or record, consisting of a linked implementation structure and a set of entry points (the head cell) consisting of pointers thereinto. For a given object specification, the implementation is constructed in several steps and three intermediate specifications are required: **SRelation**, **SGraph**, and **ImpLang**. The first two specifications depend on the given object specification whereas the third depends only on the chosen data structure. **SRelation** is the specification of the storage relations of the object specification, **SGraph** is the specification of the storage graphs of the object specification, and **ImpLang** is the specification of a toy imperative language which includes assignment, and pointers; it is our target language. The implementations are constructed stepwise by enrichment. First, the object specification is implemented by **SRelation**, then the resulting specification is implemented by **SGraph**, and then the resulting specification is implemented by **ImpLang**. The final result includes an implementation of the initial algebra of the primitive specification and it satisfies the axioms of the object specfication, but it is not necessarily an implementation of the initial algebra of the object specification.

When the implementation of **Spec** by **SGraph** requires only auxiliary operations taken from (a subset of) the operations of **Nodes&Edges'** and $\{hd_n, tl_n, hd_e, tl_e\}$, then an implementation in **ImpLang** can be synthesised which runs in constant time. If additional, recursively defined auxiliary operations are required in the implementation of **Spec** by **SGraph**, then although an implementation can be synthesised using our method, we can not guarantee that all the operations will have constant time implementations.

We have described some of the circumstances under which implementations can be constructed automatically, and also when methods such as inductive inference may be used in other circumstances.

7.2 Conclusions

We have formalised some aspects of the imperative implementation of algebraically specified ADTs.

The storage relations and storage graphs of an ADT provide us with a framework: directed graphs, in which we can explore the issues of efficient implementation. Moreover, because the storage relations and graphs are algebraically specified, we can also discuss correctness within the same framework.

Our approach is a methodology rather than an algorithm, and many aspects depend on the user; for example, the relevance of the classification of an ADT by storage type depends on the definition of useful storage types. Our approach to implementation is compilation rather than interpretation. We have found many uses for term rewriting techniques: for example, for theorem proving and inductive inference, but we have not used term rewriting for interpretation, or direct implementation. We have not always been successful when using term rewriting techniques, but are hopeful that many of the problems encountered will be overcome.

Finally, we have demonstrated the usefulness of our approach by applying various steps of the approach to several example specifications such as **`Queue`**, **`Stack`** **`Binary_Tree`**, **`Circular_List_Right`**, **`Circular_List_Left`**, **`Reversible_List`**, and **`Sequence`**.

We have not been able to consider all of the problems associated with the implementation of algebraically specified ADTs; many topics require further investigation. We outline some of these topics in the following section.

7.3 Future Work

We have plans for future work in six areas.

1) Automation

The possibilities of automating the classification of operators, the classification of an ADT by storage type, and the synthesis of implementations using the inductive theory

of the implementation have been discussed in chapters 2, 3 and 5 resp. It seems likely that the first two theorem proving tasks may be automated. The third problem requires more research; one possibility is to design and implement an inductive inference algorithm based on the results of [Lan 87].

2) Analysis of ADTs

The analysis of ADTs might be improved by considering different definitions of the storage relations and by considering a more sophisticated strategy for determining the access nodes. We could consider alternative definitions which are based on the operator classification and definitions but which also depend on other information such as priority weightings for operators and the contexts in which they appear in an algorithm. We could also investigate strategies which designate access nodes according to more sophisticated criteria. It would be interesting to compare the efficiency of the implementations synthesised using the new definitions and strategies with that of the present ones.

3) Extend the Methodology

The approach described in this thesis could be extended to include a more comprehensive treatment of data structure selection other than the present selection of linked structures for keyless and implicitly keyed ADTs. In addition, we should consider whether and how the approach can be extended to keyed ADTs; the outcome of this investigation may well depend on how the analysis can be altered to include information about operator priorities and contexts.

4) Pretty Printing

The target imperative "code" is quite remote from the usual concrete syntax of imperative programming languages; a pretty printer which maps abstract syntax on to concrete syntax, would be very helpful when presenting the results of an example implementation.

5) Extend the Specification Language and Specify Storage Relations

We should investigate further the specification language extensions which would allow the specification of storage relations as parameterised theories. For example, we should consider equation schemas and iteration over operator symbols.

6) Executable Specifications

Finally, a storage graph is none other than a non-standard representation of a (congruence class containing a) term. The choice of directed graphs as an implementation structure is well-known in the area functional programming. As storage graphs may also be compiled directly into machine code, we should investigate this further as an efficient means of executing specifications directly.

Index of Definitions

Appendix One
Algebraic Specification and Term Rewriting

Classical Equational Specification: Basic Definitions

This section introduces some algebraic concepts and definitions taken from [ADJ 78] and [EhM 85].

When S is a set, let S^* denote the free monoid of words on S with unit element λ, and subset S^+ of non-unit words.

A **signature** Σ consists of a set S of **sort** names together with an (S^*xS)-indexed family of sets of **operators** $\{\Sigma_{w,s}\}_{w\in S^*,s\in S}$. When Σ is a signature, we use S for its set of sort names; similarily for Σ' and S', Σ_0 and S_0, Σ_a and S_a, etc. When $\sigma \in \Sigma_{w,s}$, where $w \in S^+$, we say that σ is an ***s-sorted operator with arity*** w,s (or w -> s); when $\sigma \in \Sigma_{\lambda,s}$, we say that σ is an ***s-sorted constant.***

Let Σ be a signature. A **Σ-algebra** A consists of an S-indexed family of carrier sets $|A| = \{|A|_s\}_{s\in S}$ and for each operator σ with arity $s_1...s_n$->s in Σ, an operation $\sigma_A:|A|_{s1}x...x|A|_{sn}->|A|_s$.

A **Σ-term** is a well-formed term built up from the operators in Σ.

Let Σ be a signature. The ***term algebra*** of Σ, T_Σ, is constructed in the following way.

The carriers $|T_\Sigma|_s$ are inductively defined by:

i) if $\sigma \in \Sigma_{\lambda,s}$, then $\sigma \in |T_\Sigma|_s$,

ii) if $\sigma \in \Sigma_{s1...sn,s}$, and $(\forall i:1\leq i\leq n)\ t_i \in |T_\Sigma|_{si}$, then $\sigma(t_1,...,t_n)\in |T_\Sigma|_s$.

The operations are defined by:

i) if $\sigma \in \Sigma_{\lambda,s}$, then $\sigma_T := \sigma$,

ii) if $\sigma \in \Sigma_{s1...sn,s}$, and $(\forall i:1\leq i\leq n)\ t_i \in |T_\Sigma|_{si}$, then $\sigma_T(t_1,...,t_n):= \sigma(t_1,...,t_n)$.

Variables are added to signatures and Σ-terms by treating variables as constants. Let Σ be a signature and let X be an S-sorted family of sets of variables, $X = \{X_s\}$, $s \in S$. **Σ(X)** is the **enlarged** signature with the elements of X as constants; i.e.

$\Sigma(X)_{w,s} := \Sigma_{w,s}$ for $w \in S^+$,

$\Sigma(X)_{\lambda,s} := \Sigma_{\lambda,s} \cup X_s$ for $s \in S$.

Terms over the empty set of variables are called **ground terms.** $T_{\Sigma(X)}$ is denoted by $T_\Sigma(X)$ and $T_\Sigma(\{\})$ is denoted by T_Σ.

A **Σ-equation** $\forall X.\ L = R$ consists of a set X of (sorted) variables together with a pair of terms $L,R \in T_\Sigma(X)$ of the same sort.

A **specification** (Σ,E) consists of a signature Σ and a set E of Σ-equations.

Values are assigned to variables and terms containing variables. For a Σ-algebra A and S-sorted set of variables X, suppose we have a **variable assignment** function ass s.t. ass: X -> |A| and for all $s \in S$, $x_s \in X$ => $ass(x_s) \in |A|_s$. Then, ass extends uniquely to a function $\underline{ass}$ on terms with variables, $\underline{ass}$: $T_\Sigma(X)$ -> |A|, as follows:

$\underline{ass}(x) := ass(x)$ for all $x \in X$,

$\underline{ass}(f(t_1,...,t_n)) := f_A(\underline{ass}(t_1),...,\underline{ass}(t_n))$ for all $f(t_1,...,t_n) \in T_\Sigma(X)$.

A Σ-equation $\forall X.\ L = R$ is **satisfied** in a Σ-algebra A when for all variable assignments ass: X -> |A|, $\underline{ass}(L)$ and $\underline{ass}(R)$ have the same value in A.

A Σ-algebra A **satisfies** a specification (Σ,E) when every equation in E is satisfied in A. A Σ-algebra which satisfies (Σ,E) is called a ***(Σ,E)-algebra.***

A **signature morphism** is a mapping between the sorts and operator symbols of one signature to the sorts and operators symbols of another signature. Let Σ and Σ' be signatures. A signature morphism $\sigma: \Sigma \to \Sigma'$ is a pair of maps $\langle\sigma_s,\sigma_{op}\rangle$ such that for all $f \in S_{s1...sn,sm}$, $\sigma_{op}(f)$ has arity $\sigma_s(s1)...\sigma_s(sn),\sigma_s(sm)$. We usually let σ(s) and σ(f)

stand for $\sigma_s(s)$ and $\sigma_{op}(f)$ resp.

Given a signature morphism $\sigma: \Sigma \rightarrow \Sigma'$ and Σ'-algebra A', the **σ-reduct** of A' is the Σ-algebra with carrier $(|A'|_{\sigma(s)})_{s \in S}$ and operations $\sigma(f)_{A'}$ for $f \in \Sigma$.

A ***Σ-homomorphism*** h: A -> B between Σ-algebras A and B is a family of functions $(h_s: |A|_s \rightarrow |B|_s$ for $s \in S)$, when

i) for each constant $\sigma \in \Sigma_{\lambda,s}$, $h_s(\sigma_A) = \sigma_B$,

ii) the following homorphism condition holds:

$h_s(\sigma_A(t_1,...,t_n)) = \sigma_B(h_{s1}(t_1),...,h_{sn}(t_n))$,

for all $\sigma \in \Sigma_{s1...sn,s}$ and $(\forall i: 1 \leq i \leq n)\ t_i \in |A|_{si}$.

A Σ-homomorphism h: A -> B is a **Σ-isomorphism** iff there exists an inverse of h, g: B -> A, such that $gh = Id_A$ and $hg = Id_B$ (the identity functions on A and B resp.).

Given a specification (Σ,E), $\equiv_E$ is the smallest congruence on T_Σ satisfying:

1) ass(L) $\equiv_E$ ass(R) for all equations $\forall X.L = R$ and ass: $X \rightarrow T_\Sigma$,

2) $t \equiv_E t$ for all $t \in (T_\Sigma)_s$ and $s \in S$,

3) if $t \equiv_E t'$ then $t' \equiv_E t$ for all $t,t' \in (T_\Sigma)_s$ and $s \in S$,

4) if $t \equiv_E t'$ and $t' \equiv_E t''$ then $t \equiv_E t''$ for all $t,t',t'' \in (T_\Sigma)_s$ and $s \in S$,

5) if $u_i \equiv_E v_i$, for i=1,...,n, then $\sigma_T(u_1,...,u_n) \equiv_E \sigma_T(v_1,...,v_n)$ for all $\sigma: s1,...,sn \rightarrow s$, $n \geq 1$, and $u_i, v_i \in (T_\Sigma)_{si}$ for i=1,...,n.

$\equiv_E$ is called the ***congruence generated by E.***

$T_{\Sigma,E}$ denotes the term algebra T_Σ quotiented by $\equiv_E$ in the usual way. The carriers are the $(\equiv_E)_s$-congruence classes $(T_{\Sigma/\equiv E})_s$ of $|T_\Sigma|_s$, [t] denotes the $(\equiv_E)_s$-congruence class with t as member, and the operations $\sigma_{T/\equiv E}$ are defined by:

i) if $\sigma \in \Sigma_{\lambda,s}$, then $\sigma_{T/\equiv E} := [\sigma_T]$,

ii) if $\sigma \in \Sigma_{s1...sn,s}$, and $(\forall i{:}1 \leq i \leq n)\ [t_i] \in (T_{\Sigma/\equiv E})_{si}$, then

$\sigma_{T/\equiv E}([t_1],...,[t_n]) := [\sigma_T(t_1,...,t_n)]$.

Thus, every (Σ,E)-algebra A admits a unique Σ-homomorphism from $T_{\Sigma,E}$. When this is surjective, we say that A is **finitely**, or **term, generated.** Given a specification (Σ,E), $T_{\Sigma,E}$ is an **initial object** in the category of (Σ,E)-algebras and Σ-homomorphisms.

Order-Sorted Algebras: Basic Definitions

This section introduces some order-sorted algebraic definitions and results taken from [FGJ 85] and [GJM 85].

An **order-sorted signature** is a triple $(S,\leq,\Sigma)$ such that Σ is an S-sorted signature, $(S,\leq)$ is a partially-ordered set, and the operators satisfy the following **monotonicity** condition:

$(\sigma \in (\Sigma_{w,s} \cap \Sigma_{w',s'}) \wedge w \leq w') \Rightarrow s \leq s'$.

Operators may be overloaded; for each $\sigma \in \Sigma$, let T_σ be the set of **templates** or arities of σ; i.e. the set of pairs (w,s) with $\sigma \in \Sigma_{w,s}$.

An order-sorted signature Σ is **regular** when every term has a least sort and the overloaded operators with argument sorts greater than a given sort string are consistent under restriction of arguments to subsorts; i.e. Σ is regular iff for $w,w' \in S^*$ with $w \leq w'$ and given $\sigma \in \Sigma_{w',s'}$ there is a least $\langle w'',s''\rangle \in S^* \times S$ such that $\sigma \in \Sigma_{w'',s''}$ and $w \leq w''$.

An **order-sorted Σ-Algebra** A is a (Σ,E)-algebra A such that

i) if $s \leq s'$ in Σ then $|A|_s \subseteq |A|_{s'}$,

ii) if $\sigma \in (\Sigma_{s1...sn,s,s} \cap \Sigma_{s1'...sn',s'})$ with $s \leq s'$ and $s1...sn \leq s1'...sn'$, then $\sigma_A: |A|_{s1} x ... x |A|_{sn} \rightarrow |A|_s$ equals $\sigma_A: |A|_{s1'} x ... x |A|_{sn'} \rightarrow |A|_{s'}$ on $|A|_{s1} x ... x |A|_{sn}$.

For regular signatures, the usual term algebra is an initial algebra.

The logic of subsorts can be reduced to standard equational logic by treating each subsort pair $s \leq s'$ as a coercion operation from s to s' and defining a translation from the order-sorted algebra notation to the many-sorted algebra notation. There is an equivalence between the class of (regular) order-sorted algebras satisfying a set of equations and the class of standard many-sorted algebras satisfying the translation of the equations such that the initial algebras correspond under the equivalence.

Term Rewriting Systems: Basic Definitions

This section introduces some definitions and results taken from [HuO 80] and [KaS 85].

A **rewrite rule** $M \Rightarrow N$ (over $\Sigma(X)$) is an ordered pair of terms $M,N \in T_\Sigma(X)$ such that the set of variables occurring in N is a subset of the variables occurring M.

A **term rewriting system** (over $\Sigma(X)$) is a set of rewrite rules R. The **reduction relation** $\Rightarrow_R$ associated with R is the finest relation over $T_\Sigma(X)$ containing R and closed under substitution and replacement; i.e. when σ is a substitution,

$M \Rightarrow_R N$ implies $\sigma(M) \Rightarrow_R \sigma(N)$

$M \Rightarrow_R N$ implies $P[u \leftarrow M] \Rightarrow_R P[u \leftarrow N]$.

$\Rightarrow_R$ is usually referred to as $\Rightarrow$; $\Rightarrow^*$ denotes the reflexive, transitive closure of $\Rightarrow$.

A term M is irreducible, or a **normal form**, iff there is no term N such that $M \Rightarrow N$.

A term rewriting system R (over $\Sigma(X)$) is **confluent** iff for all $M,N,P \in T_\Sigma(X)$, $P \Rightarrow^* M$ and $P \Rightarrow^* N$ implies that there is some $Q \in T_\Sigma(X)$ such that $M \Rightarrow^* Q$ and $N \Rightarrow^* Q$.

A term rewriting system R (over $\Sigma(X)$) is **locally confluent** iff for all $M,N,P \in T_\Sigma(X)$, $P \Rightarrow M$ and $P \Rightarrow N$ implies that there is some $Q \in T_\Sigma(X)$ such that $M \Rightarrow^* Q$ and $N \Rightarrow^* Q$.

A term rewriting system R (over $\Sigma(X)$) is **noetherian** or **terminating** iff there is no infinite chain of reductions $M \Rightarrow M_1 \Rightarrow M_2 \Rightarrow \ldots$.

Newman's Theorem states that a terminating rewriting system R is confluent iff R is locally confluent.

Let $M \Rightarrow N$ and $M' \Rightarrow N'$ be two rules in a term rewriting system such that M and M' have no shared variables. Assume that u is a non-variable occurrence in M such that u and M' are unifiable with minimal unifier σ, then the pair of terms $\langle\sigma(M[u\leftarrow N']),\sigma(N)\rangle$ is called a **critical pair.**

The **Knuth-Bendix Theorem** states that a terminating rewriting system is locally confluent iff for every critical pair <P,Q>, the normal form of P is syntactically equivalent to the normal form of Q.

A terminating, confluent rewriting system is called **canonical.** When a term rewriting system is canonical, then every term M has a unique normal form $M\downarrow$ and $\equiv_R$ (where R is the set of bidirectional rules, or *equations*) is decidable because $M \equiv_R N$ iff $M\downarrow = N\downarrow$.

Given a signature Σ with generators Σ_g, a term rewriting system R (over Σ) is **well-spanned** iff every term of the form F(g), where F is a defined operator and g is a ground generator term, is an instance of the l.h.s. of a rewrite rule in R.

Category of Equational Specifications: Basic Definitions

This section contains some definitions concerning elementary category theory and the category of (equational) specifications taken from [EhM 85] and [Ehr 82] resp.

A **category** Cat consists of a class Obj of **objects**, for each pair $A,B \in$ Obj a set Mor(A,B) of **morphisms**, (for $f \in$ Mor(A,B) we write f: A -> B) and a composition operation

°: Mor(A,B) x Mor(B,C) -> Mor(A,C)

(f: A -> B, g: B -> C) -> (g ° f: A -> C)

such that the following axioms are satisfied:

i) $(h \circ g) \circ f = h \circ (g \circ f)$ for all morphisms f,g,h s.t. at least one side is defined,

ii) for each object $A \in Obj$ there is a morphism $id_A \in Mor(A,A)$ such that for all $f: A \to B$ and $g: C \to A$ with $B,C \in Obj$, we have $f \circ id_A = f$ and $id_A \circ g = g$.

We are interested in the category of specifications: the objects are specifications and the morphisms are signature morphisms which preserve the equational theory.

The category **spec** of specifications and specification morphisms has specifications as objects, and as morphisms from (Σ_0,E_0) to (Σ_1,E_1), those signature morphisms $\sigma: \Sigma_0 \to \Sigma_1$, for which $t \equiv_{E0} t' \Rightarrow \sigma(t) \equiv_{E1} \sigma(t')$, for ground Σ_0-terms t, t'. Such a morphism σ of **spec** is an ***embedding*** iff σ_s and σ_{op} are injective.

Let $\sigma: (\Sigma_0,E_0) \to (\Sigma_1,E_1)$ be an embedding. σ is a ***full embedding*** iff for each term $t_1 \in |T_{\Sigma 1}|_s$, for $s \in \sigma(S_0)$, there is a term $t_0 \in |T_{\Sigma 0}|$ with $\sigma(t_0) \equiv_{E1} t_1$; σ is a ***true embedding*** iff for terms $t_0,t_0' \in |T_{\Sigma 0}|$ with $\sigma(t_0) \equiv_{E1} \sigma(t_0')$, we have $t_0 \equiv_{E0} t_0'$.

Appendix Two
Example Specifications

The equations in each specification are labelled in the right hand margin. Equations which may also be regarded as left to right *rewrite rules* have labels beginning with "R"; equations which cannot be oriented have labels beginning with "E".

```
spec        Bool =
sorts       bool
gen         T   : bool
            F   : bool
ops         ∨   : bool bool -> bool
            ∧   : bool bool -> bool
            ~   : bool -> bool
            =>  : bool bool -> bool
eqns
            ~T = F                                  R1
            ~F = T                                  R2
∀b:bool.    T ∨ b = T                               R3
∀b:bool.    b ∨ T = T                               R4
∀b:bool.    F ∨ b = b                               R5
∀b:bool.    b ∨ F = b                               R6
∀b,b':bool. ~(b ∧ b') = ~b ∨ ~b'                    R7
∀b,b':bool. b => b' = ~b ∨ b'                       R8
end
```

```
spec        Nat =
basedon     Bool
sorts       nat
gen         0          : nat
            succ       : nat -> nat
ops         _=_        : nat nat ->bool
            1          : nat
            2          : nat
            3          : nat
eqns
            1 = succ(0)                             R5
            2 = succ(succ(0))                       R6
            3 = succ(succ(succ(0)))                 R7
end
```

```
spec          ANat =
basedon       Nat
ops           _+_ :nat nat-> nat
eqns
∀x,y:nat.     x + y = y + x                          E1
∀x:nat.       x + 0 = x                              R2
∀x,y:nat.     succ(x) + y = x + succ(y)              R3
end
```

```
spec          ONat =
basedon       Nat
ops           _<_ :nat nat -> bool

eqns
∀x:nat.       0 < succ(x) = T                        R1
∀x:nat.       x < x = F                              R2
∀x:nat.       x < 0 = F                              R3
∀x,y:nat      succ(x) < succ(y) = x < y              R4
end
```

```
spec          CNat =
basedon       ONat
ops           if_then_else_     :bool nat nat -> nat
eqns
∀x,y:nat.     if T then x else y = x                 R1
∀x,y:nat.     if F then x else y = y                 R2
end
```

Keyless Specifications

```
spec          Stack =
basedon       Nat
sorts         stack, nes
subsorts      nes ≤ stack
gen           create    : stack
              push      : stack nat -> nes
ops           top       : fulls -> nat
              pop       : stack  -> stack
              empty     : stack -> bool
eqns
∀d:nat,s:stack.    top(push(s,d)) =  d               R1
∀d:nat,s:stack.    pop(push(s,d)) = s                R2
                   pop(create) = create              R3
∀s:nes.            empty(s) = F                      R4
                   empty(create) = T                 R5
end
```

Σ_s={top} Σ_e={pop} Σ_o={empty} Σ_r= {}

```
spec          Stack2 =
basedon       Nat
sorts         stack, fulls
subsorts      nes ≤ stack
gen           create   : stack
              push     : stack nat -> nes
ops           top      : nes -> nat
              pop2              : stack  -> stack
              empty    : stack -> bool
eqns
∀d:nat,s:stack.     top(push(s,d)) =  d                              R1
∀d,d':nat,s:stack.  pop2(push(push(s,d),d')) = s                     R2
                    pop2(create) = create                            R3
∀d:nat.             pop2(push(create,d)) = create                    R4
∀s:nes.             empty(s) = F                                     R5
                    empty(create) = T                                R6
end
```

Σ_s={top} Σ_g={pop2} Σ_o={empty} Σ_r= {}

```
spec          Queue =
basedon       Nat
sorts         queue, neq
subsorts      neq ≤ queue
gen           eq       : queue
              add      : queue nat -> neq
ops           front    : neq -> nat
              dequeue  : queue  -> queue
              empty    : queue -> bool
eqns
∀d:nat.             front(add(eq,d)) =  d                            R1
∀q:queue,d,d':nat.  front(add(add(q,d),d')) = front(add(q,d))        R2
                    dequeue(eq) =  eq                                R3
∀d:nat.             dequeue(add(eq,d)) =  eq                         R4
∀q:queue,d,d':nat.
     dequeue(add(add(q,d),d'))=add(dequeue(add(q,d)),d')             R5
∀q:neq.             empty(q) = F                                     R6
                    empty(eq) = T                                    R7
end
```

Σ_s={front} Σ_g={dequeue} Σ_o={empty} Σ_r= {}

```
spec          List =
basedon       Nat
sorts         list, nel
subsorts      nel ≤ list
gen           nil      : list
              _:_      : nat list -> nel
ops           hd       : nel -> nat
              tl       : list  -> list
              empty    : list  -> bool
eqns
∀d:nat,l:list.   hd(d:l) =  d                          R1
∀d:nat,l:list.   tl(d:l) = l                           R2
                 tl(nil) = nil                         R3
∀l:nel.          empty(l) = F                          R4
                 empty(nil) = T                        R5
end
```

Σ_s={hd} Σ_e={tl} Σ_o={empty} Σ_r= {}

```
spec          List_app =
basedon       Nat
sorts         list, nel
subsorts      nel ≤ list
gen           nil      : list
              _:_      : nat list -> nel
ops           hd       : nel -> nat
              tl       : list  -> list
              app      : list list -> list
              empty    : list -> bool
eqns
∀d:nat,l:list.   hd(d:l) =  d                          R1
∀d:nat,l:list.   tl(d:l) = l                           R2
                 tl(nil) = nil                         R3
∀l:list.         app(l,nil) = l                        R4
∀d:nat,l,l':list. app(l,d:l') = d:(app(l,l'))          R5
∀l:nel.          empty(l) = F                          R6
                 empty(nil) = T                        R7
end
```

Σ_s={hd} Σ_e={tl} Σ_o={app,empty} Σ_r= {}

```
spec          Circular_List_right =
basedon       Nat
sorts         list, nel
subsorts      nel ≤ list
gen           nil       : list
              _:_       : nat list -> nel
ops           hd        : nel -> nat
              tl        : list  -> list
              shift     : nel -> nel
              empty     : list -> bool
eqns
∀d:nat,l:list.    hd(d:l) =  d                              R1
∀d:nat,l:list.    tl(d:l) = l                               R2
                  tl(nil) = nil                             R3
∀d:nat.           shift(d:nil) = d::nil                     R4
∀d,d':nat,l:list. shift(d:(d':l)) = d':shift(d:l)           R5
∀l:nel.           empty(l) = F                              R6
                  empty(nil) = T                            R7
end
```

Σ_S={hd} Σ_θ={tl} Σ_f={shift} Σ_O={empty}

```
spec          Circular_List_left =
basedon       Nat
sorts         list, nel
subsorts      nel ≤ list
gen           nil       : list
              _:_       : nat list -> nel
ops           hd        : nel -> nat
              last      : nel -> nat    (hidden)
              tl        : list  -> list
              rem       : nel -> list   (hidden)
              shift     : nel -> nel
              empty     : list -> bool
eqns
∀d:nat,l:list.    hd(d:l) =  d                              R1
∀d:nat,l:list.    tl(d:l) = l                               R2
                  tl(nil) = nil                             R3
∀d:nat.           last(d:nil) =  d                          R4
∀d,d':nat,l:list. last(d:(d':l)) = last(d':l)               R5
∀d:nat.           rem(d:nil) = nil                          R6
∀d,d':nat,l:list. rem(d:(d':l)) = d:rem(d':l)               R7
∀l:nel.           shift(l) = last(l):rem(l)                 E1
∀l:nel.           empty(l) = F                              R8
                  empty(nil) = T                            R9
end
```

Σ_S={hd} Σ_θ={tl} Σ_f={shift} Σ_O={empty,last,rem}

```
spec          Reversible_List_1 =
basedon       Nat
sorts         list, nel
subsorts      nel ≤ list
gen           nil      : list
              _:_      : nat list -> nel
ops           hd       : nel -> nat
              tl       : list  -> list
              app      : list list -> list   (hidden)
              rev      : list -> list
              empty    : list -> list
eqns
∀d:nat,l:list.    hd(d:l) =  d                          R1
∀d:nat,l:list.    tl(d:l) = l                           R2
                  tl(nil) = nil                         R3
∀l:list.          app(l,nil) = l                        R4
∀d:nat,l,l':list. app(l,d:l') = d:(app(l,l'))           R5
                  rev(nil) = nil                        R6
∀d:nat,l:list.    rev(d:l) = app(d:nil,rev(l))          R7
∀l:nel.           empty(l) = F                          R8
                  empty(nil) = T                        R9
end
```

Σ_s={hd} Σ_θ={tl} Σ_o={empty,app} Σ_r={rev}

```
spec          Reversible_List_2 =
basedon       Nat
sorts         list, nel
subsorts      nel ≤ list
gen           nil      : list
              _:_      : nat list -> nel
ops           hd       : nel -> nat
              tl       : list  -> list
              rev2     : list list -> list   (hidden)
              rev      : list -> list
              empty    : list -> list
eqns
∀d:nat,l:list.    hd(d:l) =  d                          R1
∀d:nat,l:list.    tl(d:l) = l                           R2
                  tl(nil) = nil                         R3
∀l:list.          rev2(l,nil) = l                       R4
∀d:nat,l,l':list. rev2(l,d:l') = rev2(d:l,l')           R5
                  rev(nil) = nil                        R6
∀d:nat,l:list.    rev(d:l) = rev2(d:nil,l)              E1
∀l:nel.           empty(l) = F                          R7
                  empty(nil) = T                        R8
end
```

Σ_s={hd} Σ_θ={tl} Σ_o={empty,rev2} Σ_r={rev}

```
spec          Binary_Tree =
basedon       ANat
sorts         tree, nonleaf
subsorts      nonleaf ≤ tree,
gen           m       : nat  -> tree
              comb    : tree tree -> nonleaf
ops           root    : tree -> nat
              left    : nonleaf -> tree
              right   : nonleaf -> tree
              leaf    : tree -> bool
eqns
∀t,t':tree.   left(comb(t,t')) = t                          R1
∀t,t':tree.   right(comb(t,t')) = t'                        R2
∀t,t':tree.   root(comb(t,t')) = root(t) + root(t')         R3
∀n:nat.       root(m(n)) = n                                R4
∀t:nonleaf.   leaf(t) = F                                   R5
∀n:nat.       leaf(m(n)) = T                                R6
end
```

Σ_g={m,comb} Σ_s={root} Σ_e={left,right} Σ_o={leaf}

```
spec          SBinary_Tree =
basedon       ANat
sorts         tree, nonleaf
subsorts      nonleaf ≤ tree,
gen           m       : nat  -> tree
              comb    : tree tree -> nonleaf
ops           root    : tree -> nat
              left    : nonleaf -> tree
              right   : nonleaf -> tree
              switch  : tree -> tree
              leaf    : tree -> bool
eqns
∀t,t':tree.   left(comb(t,t')) = t                          R1
∀t,t':tree.   right(comb(t,t')) = t'                        R2
∀t,t':tree.   root(comb(t,t')) = root(t) + root(t')         R3
∀n:nat.       root(m(n)) = n                                R4
∀n:nat.       switch(m(n)) = n                              R5
∀t,t':tree.   switch(comb(t,t')) = comb(t',t)               R6
∀t:nonleaf.   leaf(t) = F                                   R7
∀n:nat.       leaf(m(n)) = T                                R8
end
```

Σ_s={root} Σ_e={left,right} Σ_r={switch} Σ_o={leaf}

```
spec          Binary_Tree_2 =
basedon       Nat
sorts         tree, nonleaf
subsorts      nonleaf ≤ tree,
gen           m       : nat  -> tree
              comb    : tree tree nat -> nonleaf
ops           root    : tree -> nat
              left    : nonleaf -> tree
              right   : nonleaf -> tree
              leaf    : tree -> bool
eqns
∀t,t':tree,n:nat. left(comb(t,t',n)) = t                    R1
∀t,t':tree,n:nat. right(comb(t,t',n)) = t'                  R2
∀t,t':tree,n:nat. root(comb(t,t',n)) = n                    R3
∀n:nat.           root(m(n)) = n                            R4
∀t:nonleaf.       leaf(t) = F                               R7
∀n:nat.           leaf(m(n)) = T                            R8
end
```

Σ_s={root} Σ_e={left,right} Σ_r={} Σ_o={leaf}

```
spec          Sequence =
basedon       Nat
sorts         seq, nes
subsorts      nes ≤ seq
gen           eseq    : seq
              m       : nat -> nes
              conc    : nes seq -> nes
              conc    : seq nes -> nes
ops           left    : nes -> nat
              right   : nes -> nat
              lrem    : nes -> seq
              rrem    : nes -> seq
              empty   : seq -> bool
eqns
∀s:nes.       conc(s,eseq) = s                              R1
∀s:nes.       conc(eseq,s) = s                              R2
∀d:nat.       left(m(d)) = d                                R3
∀s:seq,t:nes. left(conc(t,s)) = left(t)                     R4
∀d:nat.       right(m(d)) = d                               R5
∀s:seq,t:nes. right(conc(s,t)) = right(t)                   R6
∀d:nat.       lrem(m(d)) = eseq                             R7
∀s:seq,t:nes. lrem(conc(t,s)) = conc(lrem(t),s)             R8
∀d:nat.       rrem(m(d)) = eseq                             R9
∀s:seq,t:nes. rrem(conc(s,t)) = conc(s,rrem(t))             R10
∀s:nes.       empty(s) = F                                  R11
              empty(eseq) = T                               R12
∀s,s',s":seq. conc(conc(s,s'),s") = conc(s,conc(s',s"))     E13
end
```

Σ_s={left,right} Σ_e={lrem,rrem} Σ_o={empty}

Appendix 3
Storage Type Examples

Example 3.13: Consider the **Stack** specification.

In the following, let $n \in N$, $t =_{def} \mathtt{push}^{n+1}(x_1, \ldots, x_{n+1})$ and $s =_{def} \mathtt{push}(t, y)$.

lemma 1: $\downarrow s^* = \downarrow t^* \cup \{[\mathtt{push}(t, y)]\}$.

proof:

$\downarrow t^* = \{[t],[\mathtt{push}^n(x_1, \ldots, x_n)], \ldots, [\mathtt{push}^1(x_1)], [\mathtt{create}]\}$ ($\downarrow$defn.,ind.)

$\downarrow s^* = \{[s],[t][\mathtt{push}^n(x_1, \ldots, x_n)], \ldots, [\mathtt{push}^1(x_1)], [\mathtt{create}]\}$ ($\downarrow$defn.,ind.)

$= \downarrow t^* \cup \{[s]\}$

$= \downarrow t^* \cup \{[\mathtt{push}(t, y)]\}$ □

lemma 2: $\rightarrow_s^* = \rightarrow_t^* \cup \{<[\mathtt{push}(t, y)],[t]>\}$.

proof:

$\rightarrow_t^* =$

$\{<[t],[\mathtt{push}^n(x_1, \ldots, x_n)]>, <[\mathtt{push}^n(x_1, \ldots, x_n)],[\mathtt{push}^{n-1}(x_1, \ldots, x_{n-1})]>$

$, \ldots, <[\mathtt{push}^1(x_1)],[\mathtt{create}]>\}$ ($\rightarrow$ defn.,ind.)

$\rightarrow_s^* =$

$\{<[s],[t]>, <[t],[\mathtt{push}^n(x_1, \ldots, x_n)]>, \ldots, <[\mathtt{push}^1(x_1)],[\mathtt{create}]>\}$

($\rightarrow$ defn.,ind.)

$\rightarrow_s^* = \rightarrow_t^* \cup \{<[s],[t]>\}$

$= \rightarrow_t^* \cup \{<[\mathtt{push}(t, y)],[t]>\}$ □

lemma 3: $\Downarrow s^* = \Downarrow t^* \cup \{[y]\}$.

proof:

$\Downarrow s^* = \mathtt{top}^{->}(\downarrow s^*)$ ($\Downarrow$ defn.)

$= \mathtt{top}^{->}(\downarrow t^* \cup \{[\mathtt{push}(t, y)]\}$ (lemma1)

$= \Downarrow t^* \cup \{[y]\}$ (R1,⇓ defn.)

□

lemma 4: $\Rightarrow_s^* = \Rightarrow_t^* \cup \{<[y, x_{n+1}]>\}$.

proof:

$\Rightarrow_s^* = \mathrm{top}^{->}(\rightarrow_s^*)$ (⇒ defn.)

$= \mathrm{top}^{->}(\rightarrow_t^* \cup \{<[\mathrm{push}(t, y)], [t]>\})$ (lemma 2)

$= \Rightarrow_t^* \cup \{<[y, x_{n+1}]>\}$ (R1, ⇒ defn.)

□

Example 3.14: Consider the **Queue** specification.

Recall the lemma given in §2.6 (example 2.1) which we state here again without proof.

lemma 1: $\mathrm{dequeue}(\mathrm{add}^{n+2}(x_1, \ldots, x_{n+2})) \equiv \mathrm{add}^{n+1}(x_2, \ldots, x_{n+2})$.

lemma 2: $\mathrm{front}(\mathrm{add}^{n+1}(x_1, \ldots, x_{n+1})) \equiv x_1$

proof by induction:

Base case: (n=0)

$\mathrm{front}(\mathrm{add}^1(x_1)) \equiv x_1$ (R1)

Induction step: Assume $n \in N$ and $\mathrm{front}(\mathrm{add}^{n+1}(x_1, \ldots, x_{n+1})) = x_1$.

Consider $\mathrm{front}(\mathrm{add}^{n+2}(x_1, \ldots, x_{n+1}, x_{n+1}))$.

$\mathrm{front}(\mathrm{add}^{n+2}(x_1, \ldots, x_{n+1}, x_{n+1})) \equiv \mathrm{front}(\mathrm{add}^{n+1}(x_1, \ldots, x_{n+1}))$ (R2)

$\equiv x_1$ (ass.)

Conclusion: $\mathrm{front}(\mathrm{add}^{n+1}(x_1, \ldots, x_{n+1})) \equiv x_1$. □

In the following, let $n \in N$, $t =_{def} \mathrm{add}^{n+1}(x_1, \ldots, x_{n+1})$, and $s =_{def} \mathrm{add}(t, y)$.

lemma 3: $\downarrow s^* = \{[\mathrm{add}^{n+2}(x_1, \ldots, x_{n+1}, y)], [\mathrm{add}^{n+1}(x_2, \ldots, x_{n+1}, y)],$
$\ldots, [\mathrm{add}(y)], [\mathrm{eq}]\}$.

proof by induction:

Base case: (n=0)

$\downarrow add^2(x_1, y)^* = [add^2(x_1, y)] \cup \downarrow add^1(y)^*$ (↓defn.,R5)

$= [add^2(x_1, y)] \cup [add^1(y)] \cup \downarrow eq^*$ (↓defn.,R4)

$= \{[add^2(x_1, y)], [add^1(y)], [eq]\}$ (↓defn.,R3)

Induction step: Assume $n \in N$ and

$\downarrow s^* =$

$\{[add^{n+2}(x_1, \ldots, x_{n+1}, y)], [add^{n+1}(x_2, \ldots, x_{n+1}, y)], \ldots, [add(y)], [eq]\}$.

Consider $s' =_{def} add(s, z)$.

$\downarrow s'^* = s' \cup \downarrow dequeue(s')^*$ (↓defn.)

$\downarrow s'^* =$

$\{[add^{n+3}(x_1, \ldots, x_{n+1}, y, z)]\} \cup \downarrow add^{n+2}(x_2, \ldots, x_{n+1}, y, z)^*$ (lemma1)

$= \{[add^{n+3}(x_1, \ldots, x_{n+1}, y, z)]\} \cup$

$\{[add^{n+2}(x_2, \ldots, x_{n+1}, y, z)], [add^{n+1}(x_3, \ldots, x_{n+1}, y, z)],$

$\ldots, [add^2(y, z)], [add^1(z)], [eq]\}$. (ass.)

$= \{[add^{n+3}(x_1, \ldots, x_{n+1}, y, z)], [add^{n+2}(x_2, \ldots, x_{n+1}, y, z)]$

$\ldots, [add^2(y, z)], [add(y, z)], [eq]\}$.

Conclusion:

$\downarrow s^* = \{[add^{n+2}(x_1, \ldots, x_{n+1}, y)], [add^{n+1}(x_2, \ldots, x_{n+1}, y)], \ldots, [add(y)], [eq]\}$.

□

lemma 4: $\rightarrow_s^* = \{<[add^{n+2}(x_1, \ldots, x_{n+1}, y)], [add^{n+1}(x_2, \ldots, x_{n+1}, y)]>,$

$<[add^{n+1}(x_2, \ldots, x_{n+1}, y)], [add^n(x_3, \ldots, x_{n+1}, y)]>,$

$\ldots, <[add(y)], [eq]>\}$.

proof by induction:

Base case: (n=0)

$\rightarrow add^2(x_1, y)^* = \{<[add^2(x_1, y)], [add^1(y)]>, <[add^1(y)], [eq]>\}$ (→defn.)

(R3,R4,R5)

Induction step: Assume $n \in N$ and

$$\rightarrow_s{}^* = \{<[add^{n+2}(x_1, \ldots, x_{n+1}, y)], [add^{n+1}(x_2, \ldots, x_{n+1}, y)]>,$$
$$<[add^{n+1}(x_2, \ldots, x_{n+1}, y)], [add^n(x_3, \ldots, x_{n+1}, y)]>,$$
$$\ldots, <[add(y)], [eq]>\}.$$

Consider $s' =_{def} add(s, z)$.

$\rightarrow_{s'}{}^* = \{<[s'], [add^{n+2}(x_2, \ldots, x_{n+1}, y, z)]>\} \cup \rightarrow add^{n+2}(x_2, \ldots, x_{n+1}, y, z)^*$

(→defn.)

(lemma1)

$$\rightarrow_{s'}{}^* = \{<[add^{n+3}(x_1, \ldots, x_{n+1}, y, z)], [add^{n+2}(x_2, \ldots, x_{n+1}, y, z)]>\} \cup$$
$$\{<[add^{n+2}(x_2, \ldots, x_{n+1}, y, z)], [add^{n+1}(x_3, \ldots, x_{n+1}, y, z)]>,$$
$$<[add^{n+1}(x_3, \ldots, x_{n+1}, y, z)], [add^n(x_4, \ldots, x_{n+1}, y, z)]>,$$
$$\ldots, <[add(z)], [eq]>\} \quad \text{(ass.)}$$
$$= \{<[add^{n+3}(x_1, \ldots, x_{n+1}, y, z)], [add^{n+2}(x_2, \ldots, x_{n+1}, y, z)]>,$$
$$\{<[add^{n+2}(x_2, \ldots, x_{n+1}, y, z)], [add^{n+1}(x_3, \ldots, x_{n+1}, y, z)]>,$$
$$\ldots, <[add(z)], [eq]>\}$$

Conclusion:

$\rightarrow_s{}^* =$

$$\{<[add^{n+2}(x_1, \ldots, x_{n+1}, y)], [add^{n+1}(x_2, \ldots, x_{n+1}, y)]>,$$
$$<[add^{n+1}(x_2, \ldots, x_{n+1}, y)], [add^n(x_3, \ldots, x_{n+1}, y)]>, \ldots, <[add(y)], [eq]>\}.$$

□

lemma 5: $\Downarrow s^* = \Downarrow t^* \cup \{[y]\}$.

proof:

$\Downarrow t^* = \text{front}^{->}(\downarrow t^*)$ ($\Downarrow$ defn.)

$= \text{front}^{->}\{[\text{add}^{n+1}(x_1, \ldots, x_{n+1})], [\text{add}^{n}(x_2, \ldots, x_{n+1})],$

$\ldots, [\text{add}(x_{n+1})], [eq]\}$ (lemma 2)

$= \{[x_1], [x_2], \ldots, [x_{n+1}]\}$ (R1,R2,R3)

(ind.)

$\Downarrow s^* = \text{front}^{->}(\downarrow s^*)$ ($\Downarrow$ defn.)

$= \text{front}^{->}\{[\text{add}^{n+2}(x_1, \ldots, x_{n+1}, y)], [\text{add}^{n+1}(x_2, \ldots, x_{n+1}, y)],$

$\ldots, [\text{add}(y)], [eq]\}$ (lemma 2)

$= \{[x_1], [x_2], \ldots, [x_{n+1}], [y]\}$ (R1,R2,R3)

(ind.)

$= \Downarrow t^* \cup \{[y]\}$ □

lemma 6: $\Rightarrow_s^* = \Rightarrow_t^* \cup \{<[x_{n+1}, y]>\}$.

proof:

$\Rightarrow_t^* = \text{front}^{->}(\rightarrow_t^*)$ ($\Rightarrow$ defn.)

$= \text{front}^{->}\{<[\text{add}^{n+1}(x_1, \ldots, x_{n+1})], [\text{add}^{n}(x_2, \ldots, x_{n+1})]>,$

$<[\text{add}^{n}(x_2, \ldots, x_{n+1})], [\text{add}^{n-1}(x_3, \ldots, x_{n+1})]>,$

$\ldots, <[\text{add}(x_{n+1})], [eq]>\}$ (lemma 3)

$= \{<[x_1], [x_2]>, <[x_2], [x_3]>, \ldots, <[x_n], [x_{n+1}]>\}$ (R1,R2,R3)

(ind.)

$\Rightarrow_s^* = \text{front}^{->}(\rightarrow_s^*)$ ($\Rightarrow$ defn.)

$= \text{front}^{->}(\{<[\text{add}^{n+2}(x_1, \ldots, x_{n+1}, y)], [\text{add}^{n+1}(x_2, \ldots, x_{n+1}, y)]>,$

$<[\text{add}^{n+1}(x_2, \ldots, x_{n+1}, y)], [\text{add}^{n}(x_3, \ldots, x_{n+1}, y)]>,$

$\ldots, <[\text{add}(y)], [eq]>\}$ (lemma 3)

$= \{<[x_1], [x_2]>, <[x_2], [x_3]>, \ldots, <[x_n], [x_{n+1}]>, <[x_{n+1}, y]>\}$ (R1,R2,R3)

(ind.)

$= \Rightarrow_t^* \cup \{<[x_{n+1}, y]>\}$ □

Example 3.15: Consider the **Circular_List_right** specification.
In the following, let $n \in N$.

defn1:

down $(x_{n+1}:\ldots:x_1)$ $=_{def} \{[x_{n+1}:\ldots:x_1],[x_n:\ldots:x_1],\ldots,[x_2:x_1],[x_1],[nil]\}$.

lemma 1: $shift(x_{n+1}:\ldots:x_1) \equiv x_n:\ldots:x_1:x_{n+1}$.

proof by induction:

Base case: (n=0)

$shift(x) \equiv x_1$. (R4)

Induction step: Assume $n \in N$ and $shift(x_{n+1}:\ldots:x_1) = x_n:\ldots:x_1:x_{n+1}$.

Consider $shift(x_{n+2}:\ldots:x_1)$.

$shift(x_{n+2}:\ldots:x_1) \equiv x_{n+1}:shift(x_{n+2}:x_n:\ldots:x_1)$ (R5)

$\equiv (x_{n+2}:x_n:\ldots:x_1:x_{n+2})$ (ass.)

Conclusion: $shift(x_{n+1}:\ldots:x_1) \equiv x_n:\ldots:x_1:x_{n+1}$. □

lemma 2: $\Downarrow(x_{n+1}:\ldots:x_1)^* = \{[x_{n+1}],[x_n],\ldots,[x_1]\}$

proof:

$\downarrow(x_{n+1}:\ldots:x_1)^*$

$= down(x_{n+1}:\ldots:x_1) \cup down(shift(x_{n+1}:\ldots:x_1))$ (↓ defn.)

$= down(x_{n+1}:\ldots:x_1) \cup down(x_n:\ldots:x_1:x_{n+1})$ (lemma1)

$= \{[x_{n+1}:\ldots:x_1],[x_n:\ldots:x_1],\ldots,[x_2:x_1],[x_1],[nil]\} \cup$

$\{[x_n:\ldots:x_1:x_{n+1}],[x_{n-1}:\ldots:x_1:x_{n+1}],\ldots,[x_1:x_{n+1}],[nil]\}$ (defn.1)

$\Downarrow(x_{n+1}:\ldots:x_1)^*$

$= hd^{->}(\downarrow(x_{n+1}:\ldots:x_1)^*)$ (⇓defn.)

$= hd^{->}(\{[x_{n+1}],[x_n],...,[x_2],[x_1]\} \cup \{[x_n],[x_{n-1}],...,[x_1],[x_{n+1}]\})$

$= \{[x_{n+1}:...:x_1],[x_n:...:x_1],...,[x_2:x_1],[x_1],[nil]\} \cup$

$\{[x_n:...:x_1:x_{n+1}],[x_{n-1}:...:x_1:x_{n+1}],...,[x_1:x_{n+1}],[nil]\}$ (R1,ind.)

$= \{[x_{n+1}],[x_n],...,[x_2],[x_1]\}$ □

lemma 3: $\Rightarrow(x_{n+1}:...:x_1)^* =$

$\{<[x_{n+1}],[x_n]>,<[x_n],[x_{n-1}]>,...,<[x_2],[x_1]>\} \cup \{[<[x_1],[x_{n+1}]>\}$

proof:

$\rightarrow(x_{n+1}:...:x_1)^*$

$= \{<[x_{n+1}:...:x_1],[x_n:...:x_1]>,<[x_n:...:x_1],[x_{n-1}:...:x_1]>,$

$...,<[x_2:x_1],[x_1]><[x_1],[nil]>\}$

$\cup$

$\{<[x_n:...:x_1:x_{n+1}],[x_{n-1}:...:x_1:x_{n+1}]>,$

$<[x_{n-1}:...:x_1:x_{n+1}],[x_{n-2}:...:x_1:x_{n+1}]>$

$,...,<[x_1 . x_{n+1}],[x_{n+1}]><[x_{n+1}],[nil]>\}$ (→defn)

(lemma1)

$\Rightarrow(x_{n+1}:...:x_1)^*$

$= hd^{->}(\downarrow(x_{n+1}:...:x_1)^*)$ (⇒defn.)

$= \{<[x_{n+1}],[x_n]>,<[x_n],[x_{n-1}]>,...,<[x_2],[x_1]>\} \cup$

$\{<[x_n],[x_{n-1}]>,<[x_{n-1}],[x_{n-2}]>,...,<[x_1],[x_{n+1}]>\}$ (R1,→defn.)

$= \{<[x_{n+1}],[x_n]>,<[x_n],[x_{n-1}]>,...,<[x_2],[x_1]>\} \cup \{[<[x_1],[x_{n+1}]>\}$ □

In the following, let $t =_{def} (x_{n+1}:...:x_1)$ and $s =_{def} y:t$.

lemma 4: $\Downarrow s^* = \Downarrow t^* \cup \{[y]\}$.

proof:

$\Downarrow t^* = \{[x_{n+1}],[x_n],...,[x_1]\}$ (lemma 2)

$\Downarrow s^* = \{[y],[x_{n+1}],[x_n],...,[x_1]\}$ (lemma 2)

$= \Downarrow t^* \cup \{[y]\}$ □

lemma 5: $\Rightarrow_s{}^* = (\Rightarrow_t{}^* \cup \{<[y_n], [x_{n+1}]> <[x_1], [y]>\}) \setminus \{ <[x_1], [x_{n+1}]>\}$.

$\Rightarrow_t{}^* = \{<[x_{n+1}], [x_n]>,<[x_n], [x_{n-1}]>,...,<[x_2][x_1]>\} \cup \{<[x_1][x_{n+1}]>\}$ (lemma 3)

$\Rightarrow_s{}^* = \{<[y], [x_{n+1}]>,<[x_{n+1}], [x_n]>,...,<[x_2][x_1]>\} \cup \{<[x_1][y]>\}$ (lemma 3)

$= (\Rightarrow_t{}^* \cup \{<[y_n], [x_{n+1}]> <[x_1], [y]>\}) \setminus \{ <[x_1], [x_{n+1}]>\}$ □

Example 3.16: Consider the **Circular_List_left** specification.
In the following, let $n \in N$.

defn 1:

down $(x_{n+1}:...:x_1) =_{def} \{[x_{n+1}:...:x_1],[x_n:...:x_1],...,[x_2:x_1],[x_1],[nil]\}$

lemma 1: $last(x_{n+1}:...:x_1) \equiv x_1$.

proof by induction:

Base case: (n=0)

$last(x_1) \equiv x_1$ (abbrev.,R4)

Induction step: Assume $n \in N$ and $last(x_{n+1}:...:x_1) \equiv x_1$.

Consider $last(x_{n+2}:...:x_1)$.

$last(x_{n+2}:...:x_1) \equiv last(x_{n+1}:...:x_1)$ (R5)

$\equiv x_1$ (ass.)

Conclusion: $last(x_{n+1}:...:x_1) \equiv x_1$. □

lemma2: $rem(x_{n+1}:...:x_1) \equiv (x_{n+1}:...:x_2)$.

proof by induction:

Base case: (n=0)

$rem(x_1) \equiv x_1$ (abbrev.,R6)

Induction step: Assume $n \in N$ and $\mathtt{rem}(x_{n+1}:\ldots:x_1) \equiv (x_{n+1}:\ldots:x_2)$.

Consider $\mathtt{rem}(x_{n+2}:\ldots:x_1)$.

$\mathtt{rem}(x_{n+2}:\ldots:x_1) \equiv x_{n+2}:\mathtt{rem}(x_{n+1}:\ldots:x_1)$ (R7)

$\equiv x_{n+2}:x_{n+1}:\ldots:x_2$ (ass.)

Conclusion: $\mathtt{rem}(x_{n+1}:\ldots:x_1) \equiv (x_{n+1}:\ldots:x_2)$. □

lemma 3: $\mathtt{shift}(x_{n+1}:\ldots:x_1) \equiv x_1:x_{n+1}:\ldots:x_2$.

proof:

$\mathtt{shift}(x_{n+1}:\ldots:x_1) \equiv \mathtt{last}(x_{n+1}:\ldots:x_1):\mathtt{rem}(x_{n+1}:\ldots:x_1)$ (E1)

$\equiv x_1::\mathtt{rem}(x_{n+1}:\ldots:x_1)$ (lemma1)

$\equiv x_1:x_{n+1}:\ldots:x_2$ (lemma 2)

□

lemma 4: $\Downarrow(x_{n+1}:\ldots:x_1)^* = \{[x_n],[x_{n-1}],\ldots,[x_1]\}$

proof:

$\downarrow(x_{n+1}:\ldots:x_1)^*$

$= \mathtt{down}(x_{n+1}:\ldots:x_1) \cup \mathtt{down}(\mathtt{shift}(x_{n+1}:\ldots:x_1))$ (↓ defn.)

$= \mathtt{down}(x_{n+1}:\ldots:x_1) \cup \mathtt{down}(x_1:x_{n+1}:\ldots:x_2)$ (lemma1)

$= \{[x_{n+1}:\ldots:x_1],[x_n:\ldots:x_1],\ldots,[x_2:x_1],[x_1],[\mathtt{nil}]\} \cup$

$\{[x_n:\ldots:x_1:x_{n+1}],[x_{n-1}:\ldots:x_1:x_{n+1}],\ldots,[x_1:x_{n+1}],[\mathtt{nil}]\}$ (defn.1)

$\Downarrow(x_{n+1}:\ldots:x_1)^*$

$= \mathtt{hd}^{->}(\downarrow(x_{n+1}:\ldots:x_1)^*)$ (⇓defn.)

$= \{[x_{n+1}],[x_n],\ldots,[x_2],[x_1]\} \cup \{[x_n],[x_{n-1}],\ldots,[x_1],[x_{n+1}]\}$ (↓ defn.,R1)

$= \{[x_{n+1}],[x_n],\ldots,[x_2],[x_1]\}$ □

lemma 5: $\Rightarrow(x_{n+1}:\ldots:x_1)^* =$

$\{<[x_{n+1}],[x_n]>,<[x_n],[x_{n-1}]>,\ldots,<[x_2],[x_1]>\} \cup \{<[x_1],[x_{n+1}]>\}$

proof:

$\rightarrow(x_{n+1}:\ldots:x_1)^* =$

$\{<[x_{n+1}:\ldots:x_1],[x_n:\ldots:x_1]>,<[x_n:\ldots:x_1],[x_{n-1}:\ldots:x_1]>,\ldots,<[x_2:x_1],[x_1]>,$

$<[x_1],[nil]>\}$

$\cup$

$\{<[x_1:x_{n+1}:\ldots:x_2],[x_{n+1}:\ldots:x_2]>,<[x_{n+1}:\ldots:x_2],[x_n:\ldots:x_2]>,\ldots,$

$<[x_3:x_2],[x_2]> <[x_2],[nil]>\}$ ($\rightarrow$ defn)

(lemma1)

$\Rightarrow(x_{n+1}:\ldots:x_1)^*$

$= hd^{->}(\downarrow(x_{n+1}:\ldots:x_1)^*)$ ($\Rightarrow$ defn.)

$= \{<[x_{n+1}],[x_n]>,<[x_n],[x_{n-1}]>,\ldots,<[x_2],[x_1]>\} \cup$

$\{<[x_1],[x_{n+1}]>,<[x_{n+1}],[x_n]>,\ldots,<[x_3],[x_2]>\}$ (R1, $\rightarrow$ defn.)

$= \{<[x_{n+1}],[x_n]>,<[x_n],[x_{n-1}]>,\ldots,<[x_2],[x_1]>\} \cup \{<[x_1],[x_{n+1}]>\}$ $\square$

In the following, let $t =_{def} (x_{n+1}:\ldots:x_1)$ and $s =_{def} y:t$.

lemma 6: $\Downarrow s^* = \Downarrow t^* \cup \{[y]\}$.

proof:

$\Downarrow t^* = \{[x_{n+1}],[x_n],\ldots,[x_1]\}$ (lemma 4)

$\Downarrow s^* = \{[y],[x_{n+1}],[x_n],\ldots,[x_1]\}$ (lemma 4)

$= \Downarrow t^* \cup \{[y]\}$ $\square$

lemma7: $\Rightarrow_s^* = (\Rightarrow_t^* \cup \{<[y_n],[x_{n+1}]> <[x_1],[y]>\}) \setminus \{<[x_1],[x_{n+1}]>\}$.

proof:

$\Rightarrow_t^* = \{<[x_{n+1}], [x_n]>,<[x_n], [x_{n-1}]>,...,<[x_2][x_1]>\} \cup \{<[x_1],[x_{n+1}]>\}$ (lemma 5)

$\Rightarrow_s^* = \{<[y], [x_{n+1}]>,<[x_{n+1}], [x_n]>,...,<[x_2][x_1]>\} \cup \{<[x_1],[y]>\}$ (lemma 5)

$= (\Rightarrow_t^* \cup \{<[y_n], [x_{n+1}]> <[x_1], [y]>\}) \setminus \{<[x_1], [x_{n+1}]>\}$ □

Example 3.17: Consider the **Reversible_List_1** specification.
In the following, let $n \in N$.

defn 1:

down $(x_{n+1}:...:x_1) =_{def} \{[x_{n+1}:...:x_1],[x_n:...:x_1],...,[x_2:x_1],[x_1],[nil]\}$

lemma 1: $(\forall d:nat)$ $app(d:nil,x_{n+1}:...:x_1) \equiv x_{n+1}:...:x_1:d.$

proof by induction:

Base case: (n=0)

$app(d:nil,x_1) \equiv x_1:app(d:nil,nil)$ (R6)

$\equiv x_1:d:nil$ (R5)

$\equiv x_1:d$

Induction step: Assume $n \in N$ and $app(d:nil,x_{n+1}:...:x_1) \equiv x_{n+1}:...:x_1:d.$

Consider $app(d:nil, x_{n+2}:x_{n+1}:...:x_1)$.

$app(d:nil, x_{n+2}:x_{n+1}:...:x_1)$

$\equiv x_{n+2}:app(d:nil, x_{n+1}:...:x_1)$ (R6)

$\equiv x_{n+2}:x_{n+1}:...:x_1;d)$ (ass.)

Conclusion: $app(d:nil,x_{n+1}:...:x_1) \equiv x_{n+1}:...:x_1:d.$ □

lemma 2: $rev(x_{n+1}:...:x_1) \equiv (x_1:...:x_{n+1}).$

proof by induction:

Base case: (n=0)

$rev(x_1) \equiv app(x_1:nil,rev(nil))$ (R8)

$\equiv$ app (x_1:nil, nil) (R7)

$\equiv$ x_1:nil

Induction step: Assume n $\in$ N and $\mathtt{rev}(x_{n+1}:\ldots:x_1) \equiv (x_1:\ldots:x_{n+1})$.

Consider $\mathtt{rev}(x_{n+2}:\ldots:x_1)$.

$\mathtt{rev}(x_{n+2}:\ldots:x_1)$ $\equiv$ app (x_{n+2}:nil,rev($x_{n+1}:\ldots:x_1$)) (R8)

$\equiv$ app (x_{n+2}:nil, $x_1:\ldots:x_{n+1}$) (ass.)

$\equiv$ $x_1:\ldots:x_{n+1}:x_{n+2}$ (lemma 1)

Conclusion: $\mathtt{rev}(x_{n+1}:\ldots:x_1) \equiv (x_1:\ldots:x_{n+1})$. □

In the following, let $t =_{def} (x_{n+1}:\ldots:x_1)$ and $s =_{def} y:t$.

lemma 3: $\Downarrow(x_{n+1}:\ldots:x_1)^* = \{[x_{n+1}],[x_n],\ldots,[x_1]\}$.

proof:

$\downarrow(x_{n+1}:\ldots:x_1)^*$

= down ($x_{n+1}:\ldots:x_1$) $\cup$ down (rev ($x_{n+1}:\ldots:x_1$)) (↓ defn.)

= down ($x_{n+1}:\ldots:x_1$) $\cup$ down ($x_1:\ldots:x_{n+1}$) (lemma 2)

= $\{[x_{n+1}:\ldots:x_1],[x_n:\ldots:x_1],\ldots,[x_2:x_1],[x_1],[\mathtt{nil}]\} \cup$

$\{[x_1:\ldots:x_{n+1}],[x_2:\ldots:x_{n+1}],\ldots,[x_n:x_{n+1}],[\mathtt{nil}]\}$ (defn.1)

$\Downarrow(x_{n+1}:\ldots:x_1)^*$

= $\mathrm{hd}^{->}(\downarrow x_{n+1}:\ldots:x_1)^*$ (⇓defn.)

= $\mathrm{hd}^{->}(\{[x_{n+1}:\ldots:x_1],[x_n:\ldots:x_1],\ldots,[x_2:x_1],[x_1],[\mathtt{nil}]\}$

$\cup \{[x_1:\ldots:x_{n+1}],[x_2:\ldots:x_{n+1}],\ldots,[x_n:x_{n+1}],[\mathtt{nil}]\})$

= $\{[x_{n+1}],[x_n],\ldots,[x_1]\}$ (R1,ind.)

□

lemma 4: $\Rightarrow(x_{n+1}:\ldots:x_1)^* = \{<[x_i],[x_{i-1}]>,<[x_j],[x_{j+1}]> \mid 2\leq i\leq n+1,\ 1\leq j\leq n\}$

proof:

$\rightarrow(x_{n+1}:\ldots:x_1)^* =$

$\{<[x_{n+1}:\ldots:x_1],[x_n:\ldots:x_1]>,<[x_n:\ldots:x_1],[x_{n-1}:\ldots:x_1]>,\ldots,<[x_2:x_1],[x_1]>,$

$<[x_1],[nil]>\}$

$\cup$

$\{<[x_1:\ldots:x_{n+1}],[x_2:\ldots:x_{n+1}]>,<[x_2:\ldots:x_{n+1}],[x_3:\ldots:x_{n+1}]>,$

$\ldots,<[x_n:x_{n+1}],[x_{n+1}]><[x_{n+1}],[nil]>\}$ ($\rightarrow$ defn)

(lemma1)

$\Rightarrow(x_{n+1}:\ldots:x_1)^*$

$= hd^{->}(\rightarrow(x_{n+1}:\ldots:x_1)^*)$ ($\Rightarrow$ defn.)

$= \{<[x_{n+1}],[x_n]>,<[x_n],[x_{n-1}]>,\ldots,<[x_2],[x_1]>\} \cup$

$\{<[x_1],[x_2]>,<[x_2],[x_3]>,\ldots,<[x_n],[x_{n+1}]>\}$ (R1,$\rightarrow$ defn.)

$= \{<[x_i],[x_{i-1}]>,<[x_j],[x_{j+1}]> \mid 2\leq i\leq n+1,\ 1\leq j\leq n\}$ □

In the following, let $t=_{def}(x_{n+1}:\ldots:x_1)$, $s=_{def} y:t$ and $x_{n+2} =_{def} y$.

lemma 5: $\Downarrow s^* = \Downarrow t^* \cup \{[y]\}$.

proof:

$\Downarrow t^* = \{[x_{n+1}],[x_n],\ldots,[x_1]\}$ (lemma 3)

$\Downarrow s^* = \{[y],[x_{n+1}],[x_n],\ldots,[x_1]\}$ (lemma 3)

$= \Downarrow t^* \cup \{[y]\}$ □

lemma 6: $\Rightarrow_s{}^* = \Rightarrow_t{}^* \cup \{<[y],[x_{n+1}]>,<[x_{n+1}],[y]>\}$.

proof:

$\Rightarrow_t{}^* = \{<[x_i],[x_{i-1}]>,<[x_j],[x_{j+1}]> \mid 2\leq i\leq n+1,\ 1\leq j\leq n\}$ (lemma 4)

$\Rightarrow_s{}^* = \{<[x_i],[x_{i-1}]>,<[x_j],[x_{j+1}]> \mid 2\leq i\leq n+2,\ 1\leq j\leq n+1\}$ (lemma 4)

$= \Rightarrow_t{}^* \cup \{<[y],[x_{n+1}]>,<[x_{n+1}],[y]>\}$ □

Example 3.18: Consider the **Sequence** specification.

In the following, let $n \in N$.

lemma 1: $\mathtt{lrem}(\mathtt{sq}(x_1:\ldots:x_{n+2})) \equiv \mathtt{sq}(x_2:\ldots:x_{n+2})$

proof:

$\mathtt{lrem}(\mathtt{sq}(x_1:\ldots:x_{n+2})) \equiv \mathtt{lrem}(\mathtt{conc}(\mathtt{m}(x_1),\mathtt{sq}(x_2:\ldots:x_{n+2}))$ (abbrev.)

$\equiv \mathtt{conc}(\mathtt{lrem}(\mathtt{m}(x_1),\mathtt{sq}(x_2:\ldots:x_{n+2}))$ (R8)

$\equiv \mathtt{conc}(\mathtt{eseq},\mathtt{sq}(x_2:\ldots:x_{n+2}))$ (R7)

$\equiv \mathtt{sq}(x_2:\ldots:x_{n+2})$ (R2)

□

lemma 2: $\mathtt{rrem}(\mathtt{sq}(x_1:\ldots:x_{n+2})) \equiv \mathtt{sq}(x_1:\ldots:x_{n+1})$.

proof by induction:

Base case: (n=0)

$\mathtt{rrem}(\mathtt{sq}(x_1,x_2)) \equiv \mathtt{rrem}(\mathtt{conc}(\mathtt{m}(x_1),\mathtt{m}(x_2)))$ (abbrev.)

$\equiv \mathtt{conc}(\mathtt{m}(x_1),\mathtt{rrem}(\mathtt{m}(x_2)))$ (R10)

$\equiv \mathtt{conc}(\mathtt{m}(x_1),\mathtt{eseq})$ (R9)

$\equiv \mathtt{sq}(x_1)$ (abbrev.)

Induction Step: Assume $n \in N$ and $\mathtt{rrem}(\mathtt{sq}(x_1:\ldots:x_{n+2})) \equiv \mathtt{sq}(x_1:\ldots:x_{n+1})$.

Consider $\mathtt{rrem}(\mathtt{sq}(x_1:\ldots:x_{n+3}))$.

$\mathtt{rrem}(\mathtt{sq}(x_1:\ldots:x_{n+3}))$

$\equiv \mathtt{rrem}(\mathtt{conc}(\mathtt{m}(x_1),\mathtt{sq}(x_2:\ldots:x_{n+3}))$ (abbrev.)

$\equiv \mathtt{conc}(\mathtt{m}(x_1),\mathtt{rrem}(\mathtt{sq}(x_2:\ldots:x_{n+3}))$ (R10)

$\equiv \mathtt{conc}(\mathtt{m}(x_1),\mathtt{rrem}(\mathtt{sq}(x_2:\ldots:x_{n+2}))$ (ass.)

$\equiv \mathtt{sq}(x_1:\ldots:x_{n+2})$ (abbrev.)

Conclusion: $\mathtt{rrem}(\mathtt{sq}(x:\ldots:x_{n+2})) \equiv \mathtt{sq}(x_1:\ldots:x_{n+1})$. □

lemma 3: $\mathtt{left}(\mathtt{sq}(x_1:\ldots:x_{n+1})) \equiv x_1$

proof:

$\mathtt{left}(\mathtt{sq}(x_1:\ldots:x_{n+2}))$

$\equiv \mathtt{left}(\mathtt{conc}(\mathtt{m}(x_1),\mathtt{sq}(x_2:\ldots:x_{n+2}))$ (abbrev.)

$\equiv \mathtt{left}(\mathtt{m}(x_1))$ (R4)

$\equiv x_1$ (R3)

□

lemma 4: $\mathtt{right}(\mathtt{sq}(x_1:\ldots:x_{n+1})) \equiv x_{n+1}$.

proof by induction:

Base case: (n=0)

$\mathtt{right}(\mathtt{sq}(x_1)) \equiv \mathtt{right}(\mathtt{m}(x_1))$ (abbrev.)

$\equiv x_1$ (R5)

Induction Step: Assume $n \in N$ and $\mathtt{right}(\mathtt{sq}(x_1:\ldots:x_{n+1})) \equiv x_{n+1}$.

Consider $\mathtt{right}(\mathtt{sq}(x_1:\ldots:x_{n+2}))$.

$\mathtt{right}(\mathtt{sq}(x_1:\ldots:x_{n+2}))$

$\equiv \mathtt{right}(\mathtt{conc}(\mathtt{m}(x_1),\mathtt{sq}(x_2:\ldots:x_{n+2})))$ (abbrev.)

$\equiv \mathtt{right}(\mathtt{sq}(x_2:\ldots:x_{n+2}))$ (R6)

$\equiv x_{n+2}$ (ass.)

Conclusion: $\mathtt{right}(\mathtt{sq}(x_1:\ldots:x_{n+1})) \equiv x_{n+1}$. □

lemma 5: $\downarrow\mathtt{sq}(x_1:\ldots:x_{n+1})^* = \{[\mathtt{sq}(x_i:\ldots:x_j)] \mid 1 \le i \le j \le n+1\} \cup \{[\mathtt{eseq}]\}$.

proof by induction:

Base case: (n=0)

$\downarrow\mathtt{sq}(x_1)^* = \{[\mathtt{sq}(x_1)]\} \cup \downarrow\mathtt{eseq}$ (↓defn.)

$= \{[sq(x_1)]\} \cup \{[eseq]\}$ (↓defn.)

Induction Step: Assume $n \in N$ and

$$\downarrow sq(x_1:\ldots:x_{n+1})^* = \{[sq(x_i:\ldots:x_j)] | 1 \leq i \leq j \leq n+1\} \cup \{[eseq]\}.$$

Consider $\downarrow sq(x_1:\ldots:x_{n+2})^*$.

$\downarrow sq(x_1:\ldots:x_{n+2})^*$

$= \{[sq(x_2:\ldots:x_{n+2})]\} \cup \downarrow lrem(sq(x_1:\ldots:x_{n+2}))^*$

$\cup \downarrow rrem(sq(x_1:\ldots:x_{n+2}))^*$ (↓defn.)

$= \{[sq(x_2:\ldots:x_{n+2})]\} \cup \downarrow sq(x_2:\ldots:x_{n+2})^* \cup \downarrow sq(x_1:\ldots:x_{n+1})^*$ (lemma 1,2)

$= \{[sq(x_2:\ldots:x_{n+2})]\} \cup \{[eseq]\} \cup \{[sq(x_i:\ldots:x_j)] | 2 \leq i \leq j \leq n+2\}$

$\cup \{[sq(x_i:\ldots:x_j)] | 1 \leq i \leq j \leq n+1\}$ (ass.)

$= \{[sq(x_1:\ldots:x_{n+2})] | 1 \leq i \leq j \leq n+1\} \cup \{[eseq]\}$

Conclusion: $\downarrow sq(x_1:\ldots:x_{n+1})^* = \{[sq(x_1:\ldots:x_{n+1})] | 1 \leq i \leq j \leq n+1\} \cup \{[eseq]\}$.

□

<u>lemma 6</u>: $\Downarrow sq(x_1:\ldots:x_{n+1})^* = \{[x_1],\ldots,[x_{n+1}]\}$.

<u>proof:</u>

$\Downarrow sq(x_1:\ldots:x_{n+1})^*$

$= left^{->}(\downarrow sq(x_1:\ldots:x_{n+1})^*) \cup left^{->}(\downarrow sq(x_1:\ldots:x_{n+1})^*)$(⇒defn.)

$= left^{->}(\downarrow sq(x_1:\ldots:x_{n+1})^*) \cup left^{->}(\downarrow sq(x_1:\ldots:x_{n+1})^*)$(⇒defn.)

$= left^{->}(\{[sq(x_1:\ldots:x_{n+1})] | 1 \leq i \leq j \leq n+1\} \cup \{[eseq]\})$

$\cup\ right^{->}\{[sq(x_1:\ldots:x_{n+1})] | 1 \leq i \leq j \leq n+1\} \cup \{[eseq]\}$ (lemma 5)

$= \{[x_i] | 1 \leq i \leq n+1\} \cup \{[x_j] | 1 \leq j \leq n+1\}$ (lemma3,4)

$= \{[x_1],\ldots,[x_{n+1}]\}$

lemma 7: $\rightarrow sq(x_1:\ldots:x_{n+1})^* =$

$\{<[sq(x_i:\ldots:x_j)],[sq(x_i:\ldots:x_{j-1})]>,<[sq(x_i:\ldots:x_j)],[sq(x_{i+1}:\ldots:x_j)]>$

$\quad | 1 \leq i < j \leq n+1\} \cup \{<[sq(x_i)],[eseq]> | 1 \leq i \leq n+1\}.$

proof by induction:

Base case: (n=0)

$\rightarrow sq(x_1)^*$

$= \{<[sq(x_1)],[lrem(sq(x_1))]>,<[sq(x_1)],[rrem(sq(x_1))]>\}$

$\quad \cup \rightarrow lrem(sq(x_1))^* \cup \rightarrow rrem(sq(x_1))^*$ (→defn.)

$= \{<[sq(x_1)],[eseq]>\} \cup \rightarrow eseq$ (R7, R9)

$= \{<[sq(x_1)],[eseq]>\}$ (→defn.)

Induction Step: Assume $n \in N$ and

$\rightarrow sq(x_1:\ldots:x_{n+1})^* =$

$\{<[sq(x_i:\ldots:x_j)],[sq(x_i:\ldots:x_{j-1})]>,<[sq(x_i:\ldots:x_j)],[sq(x_{i+1}:\ldots:x_j)]>$

$\quad | 1 \leq i < j \leq n+1\} \cup \{<[sq(x_i)],[eseq]> | 1 \leq i \leq n+1\}$

Consider y where $y =_{def} sq(x_1:\ldots:x_{n+2})$.

$\rightarrow y^*$

$= \{<[y],[lrem(y)]>,<[y],[rrem(y)]>\} \cup \rightarrow lrem(y)^* \cup \rightarrow rrem(y)^*$ (→defn.)

$= \{<[y],[sq(x_2:\ldots:x_{n+2})]>,<[y],[sq(x_1:\ldots:x_{n+1})]>\}$

$\quad \cup \rightarrow sq(x_2:\ldots:x_{n+2})^* \cup \rightarrow sq(x_1:\ldots:x_{n+1})^*$ (lemma 1,2)

$= \{<[y],[sq(x_2:\ldots:x_{n+2})]>,<[y],[sq(x_1:\ldots:x_{n+1})]>\} \cup$

$\{<[sq(x_i:\ldots:x_j)],[sq(x_i:\ldots:x_{j-1})]>,<[sq(x_i:\ldots:x_j)],[sq(x_{i+1}:\ldots:x_j)]>$

$\quad | 2 \leq i < j \leq n+2\} \cup \{<[sq(x_i)],[eseq]> | 2 \leq i \leq n+2\}$

$\quad \cup$

$\{<[sq(x_i:\ldots:x_j)],[sq(x_i:\ldots:x_{j-1})]>,<[sq(x_i:\ldots:x_j)],[sq(x_{i+1}:\ldots:x_j)]>$

$\quad | 1 \leq i < j \leq n+1\} \cup \{<[sq(x_i)],[eseq]> | 1 \leq i \leq n+1\}$ (ass.)

=

$\{<[sq(x_i:\ldots:x_j)],[sq(x_i:\ldots:x_{j-1})]>,<[sq(x_i:\ldots:x_j)],[sq(x_{i+1}:\ldots:x_j)]>$

$\quad | \ 1 \leq i < j \leq n+2\} \cup \{<[sq(x_i)],[eseq]> | \ 1 \leq i \leq n+2\}$

Conclusion $\rightarrow sq(x_1:\ldots:x_{n+1})^* =$

$\{<[sq(x_i:\ldots:x_j)],[sq(x_i:\ldots:x_{j-1})]>,<[sq(x_i:\ldots:x_j)],[sq(x_{i+1}:\ldots:x_j)]>$

$\quad | \ 1 \leq i < j \leq n+1\} \cup \{<[sq(x_i)],[eseq]> | 1 \leq i \leq n+1\}.$ □

lemma 8: $\Rightarrow sq(x_1:\ldots:x_{n+1})^* =$

$\{<[x_i],[x_i]> | \ 1 \leq i \leq n+1\} \cup \{<[x_i],[x_{i+1}]> | \ 1 \leq i < n+1\} \cup \{<[x_j],[x_{j-1}]> | \ 1 < j \leq n+1\}.$

proof:

$\Rightarrow sq(x_1:\ldots:x_{n+1})^*$

$= left^{->}(\rightarrow(x_1:\ldots:x_{n+1})^*) \cup right^{->}(\rightarrow(x_1:\ldots:x_{n+1})^*)$ ($\Rightarrow$ defn.)

$= left^{->}(\{<[sq(x_i:\ldots:x_j)],[sq(x_i:\ldots:x_{j-1})]>,$

$\quad <[sq(x_i:\ldots:x_j)],[sq(x_{i+1}:\ldots:x_j)]>$

$\quad | \ 1 \leq i < j \leq n+1\} \cup \{<[sq(x_i)],[eseq]> | \ 1 \leq i \leq n+1\})$

$\cup$

$right^{->}(\{<[sq(x_i:\ldots:x_j)],[sq(x_i:\ldots:x_{j-1})]>,$

$\quad <[sq(x_i:\ldots:x_j)],[sq(x_{i+1}:\ldots:x_j)]>$

$\quad | \ 1 \leq i < j \leq n+1\} \cup \{<[sq(x_i)],[eseq]> | \ 1 \leq i \leq n+1\})$ (lemma 7)

$= \{<[sq(x_i)],[sq(x_i)]>,<[sq(x_i)],[sq(x_{i+1})]> | 1 \leq i < n+1\} \cup$

$\{<[sq(x_j)],[sq(x_{j-1})]>,<[sq(x_j)],[sq(x_j)]> | 1 \leq j \leq n+1\}$ (lemma 3,4)

$= \{<[x_i],[x_i]> | \ 1 \leq i \leq n+1\} \cup \{<[x_i],[x_{i+1}]> | \ 1 \leq i < n+1\} \cup \{<[x_j],[x_{j-1}]> | \ 1 < j \leq n+1\}$

□

In the following, let t =def $sq(x_1:\ldots:x_{n+1})$ and s =def $sq(x_1:\ldots:x_{n+2})$.

lemma 9: $\Downarrow s^* = \Downarrow t^* \cup \{[x_{n+2}]\}$.

proof:

$\Downarrow t^* = \{[x_1],\ldots,[x_{n+1}]\}$ (lemma 6)

$\Downarrow s^* = \{[x_1],\ldots,[x_{n+2}]\}$ (lemma 6)

$= \Downarrow t^* \cup \{[x_1]\}$ □

lemma 10: $\Rightarrow s^* = \Rightarrow t^* \cup \{<[x_{n+2}],[x_{n+2}]>\} \cup \{<[x_{n+2}],[x_{n+1}]>, <[x_{n+1}],[x_{n+2}]>\}$.

proof:

$\Downarrow t^*$

$= \{<[x_i],[x_i]>|\ 1\leq i\leq n+1\} \cup \{<[x_i],[x_{i+1}]>|\ 1\leq i<n+1\} \cup$

$\{<[x_j],[x_{j-1}]>|\ 1<j\leq n+1\}$. (lemma 8)

$\Downarrow s^*$

$= \{<[x_i],[x_i]>|\ 1\leq i\leq n+2\} \cup \{<[x_i],[x_{i+1}]>|\ 1\leq i<n+2\} \cup$

$\{<[x_j],[x_{j-1}]>|\ 1<j\leq n+2\}$. (lemma 8)

$= \Rightarrow t^* \cup \{<[x_{n+2}],[x_{n+2}]>\} \cup \{<[x_{n+2}],[x_{n+1}]>, <[x_{n+1}],[x_{n+2}]>\}$. □

```
/*----------------------------STANDARD KNUTH-BENDIX CONFIGURATION---
/*list_app rewrite rules - KB complete*/
~(T) = F
~(F) = T
T V b = T
b V T = T
F V b = b
b V F = b
T&b = b
b&T = b
F&b = F
b&F = F
~((b&b1)) = ~(b)V~(b1)
z=z = T
b=>b1 = ~(b)V b1
1 = succ(0)
2 = succ(succ(0))
0=succ(n) = F
succ(n)=0 = F
3 = succ(succ(succ(0)))
succ(n)=succ(n1) = n=n1
4 = succ(succ(succ(succ(0))))
tl(nil) = nil
empty(nil) = T
app(nil,l) = l
app(l,nil) = l
hd((n++l)) = n
tl((n++l)) = l
app(l,(n++l1)) = n++app(l,l1)
```

```
/*-------------------------------STANDARD KNUTH-BENDIX CONFIGURATION-----
/*stack rewrite rules - KB complete*/
~(T) = F
~(F) = T
T V b = T
b V T = T
F V b = b
b V F = b
T&b = b
b&T = b
F&b = F
b&F = F
~((b&b1)) = ~(b)V~(b1)
z=z = T
b=>b1 = ~(b)V b1
1 = succ(0)
2 = succ(succ(0))
0=succ(n) = F
succ(n)=0 = F
3 = succ(succ(succ(0)))
succ(n)=succ(n1) = n=n1
4 = succ(succ(succ(succ(0))))
empty(f) = F
empty(create) = T
pop(create) = create
top(push(s,n)) = n
pop(push(s,n)) = s
```

```
/*-------------------------------STANDARD KNUTH-BENDIX CONFIGURATION-----
/*queue rewrite rules - KB complete*/
~(T) = F
~(F) = T
T V b = T
b V T = T
F V b = b
b V F = b
T&b = b
b&T = b
F&b = F
b&F = F
~((b&b1)) = ~(b)V~(b1)
z=z = T
b=>b1 = ~(b)V b1
1 = succ(0)
2 = succ(succ(0))
0=succ(n) = F
succ(n)=0 = F
3 = succ(succ(succ(0)))
succ(n)=succ(n1) = n=n1
4 = succ(succ(succ(succ(0))))
empty(f) = F
empty(eq) = T
dequeue(eq) = eq
dequeue(add(eq,n)) = eq
dequeue(add(add(q,n),n1)) = add(dequeue(add(q,n)),n1)
front(add(eq,n)) = n
front(add(add(q,n),n1)) = front(add(q,n))
```

```
/*------------------------- Standard KNUTH-BENDIX Configuration ---
/*sequence rewrite rules - userKBO - KB complete*/
~(T) = F
~(F) = T
1 = succ(0)
empty(f) = F
empty(eseq) = T
T V b = T
b V T = T
F V b = b
b V F = b
T&b = b
b&T = b
F&b = F
b&F = F
2 = succ(succ(0))
conc(s,eseq) = s
conc(eseq,s) = s
z=z = T
left(m(n)) = n
right(m(n)) = n
lrem(m(n)) = eseq
rrem(m(n)) = eseq
left(conc(f,s)) = left(f)
right(conc(s,f)) = right(f)
rrem(conc(s,f)) = conc(s,rrem(f))
lrem(conc(f,s)) = conc(lrem(f),s)
0=succ(n) = F
succ(n)=0 = F
3 = succ(succ(succ(0)))
~((b&b1)) = ~(b)V~(b1)
b=>b1 = ~(b)V~(b1)
4 = succ(succ(succ(succ(0))))
succ(n)=succ(n1) = n=n1
```

```
/*---------------------- Default KNUTH-BENDIX Configuration ----
/*Circular_List_right rewrite rules - fixedKBO - KB complete*/
~(T) = F
~(F) = T
1 = succ(0)
tl(nil) = nil
empty(nil) = T
empty(f) = F
T V b = T
b V T = T
F V b = b
b V F = b
T&b = b
b&T = b
F&b = F
b&F = F
x=x = T
2 = succ(succ(0))
b=>b1 = ~(b)V b1
3 = succ(succ(succ(0)))
shift((n++nil)) = n++nil
hd((n++l)) = n
tl((n++l)) = l
~((b&b1)) = ~(b)V~(b1)
4 = succ(succ(succ(succ(0))))
shift((n1++(n++l))) = n++shift((n1++l))
```

```
/*------------------------ Default KNUTH-APENDIX Configuration
/*Circular_List_left  - user.KB0 -  KB complete*/
~(T) = F
~(F) = T
1 = succ(0)
hd(nil) = nil
empty(nil) = T
empty(f) = F
T V b = T
b V T = T
F V b = b
b V F = b
T&b = b
b&T = b
F&b = F
b&F = F
2 = succ(succ(0))
z=z = T
b=>b1 = ~(b)V b1
3 = succ(succ(succ(0)))
hd((n++l)) = n
tl((n++l)) = l
last((n++nil)) = n
last((n++f)) = last(f)
rem((n++nil)) = nil
rem((n++f)) = n++rem(f)
~((b&b1)) = ~(b)V~(b1)
4 = succ(succ(succ(succ(0))))
shift(f) = last(f)++rem(f)
```

```
/*-------------------------------STANDARD KNUTH-BENDIX CONFIGURATION---
/*rev_list_1 rewrite rules - KB complete using inductive order
~(T) = F
~(F) = T
T V b = T
b V T = T
F V b = b
b V F = b
T&b = b
b&T = b
F&b = F
b&F = F
~((b&b1)) = ~(b)V~(b1)
(z=z) = T
b=>b1 = ~(b)V b1
1 = (succ(0))
2 = (succ(succ(0)))
(0=succ(n)) = F
(succ(n)=0 )= F
3 = (succ(succ(succ(0))))
(succ(n)=succ(n1)) = n=n1
4 = (succ(succ(succ(succ(0)))))
empty(f) = F
tl(nil) = nil
empty(nil) = T
app(nil,l) = l
app(l,nil) = l
hd(n ++ l) = n
tl(n ++ l) = l
app(l,(n++l1)) = n++app(l,l1)
rev(nil) = nil
rev(n++l) = app((n++nil),rev(l))
```

```
/*----------------------------Standard KNUTH-BENDIX Configuration-----
/*reversible_list_2  -userK30  - KB complete*/
~(T) = F
~(F) = T
1 = succ(0)
empty(f) = F
tl(nil) = nil
empty(nil) = T
rev(nil) = nil
T V b = T
b V T = T
F V b = b
b V F = b
T&b = b
b&T = b
b&F = F
z=z = T
2 = succ(succ(0))
F&b = F
rev2(nil,l) = l
b=>bl = ~(b)V bl
0=succ(n) = F
succ(n)=0 = F
3 = succ(succ(succ(0)))
hd((n++l)) = n
tl((n++l)) = l
~((b&bl)) = ~(b)V~(bl)
succ(n)=succ(n1) = n=n1
4 = succ(succ(succ(succ(0))))
rev2((n++l),l1) = rev2(l,(n++l1))
rev((n++l)) = rev2(l,(n++nil))
```

```
/*-------------------------STANDARD KNUTH-BENDIX CONFIGURATIO
/*stack2 rewrite rules - KB complete*/
~(T) = F
~(F) = T
T V b = T
b V T = T
F V b = b
b V F = b
T&b = b
b&T = b
F&b = F
b&F = F
~((b&b1)) = ~(b)V~(b1)
z=z = T
b=>b1 = ~(b)V b1
1 = succ(0)
2 = succ(succ(0))
0=succ(n) = F
succ(n)=0 = F
3 = succ(succ(succ(0)))
succ(n)=succ(n1) = n=n1
4 = succ(succ(succ(succ(0))))
empty(f) = F
empty(create) = T
pop2(create) = create
top(push(s,n)) = n
pop2(push(create,n)) = create
pop2(push(push(s,n),n1)) = s
```

References

[ADJ 78]

J.A.Goguen, J.W.Thatcher, E.G.Wagner, An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types, *Current Trends in Programming Methodology*, Number 4, Chapter 5, 1978.

[AHU 83]

A.V.Aho, J.E.Hopcroft, J.D.Ullman, *Data Structures and Algorithms*, Addison-Wesley 1983.

[Bar 83]

J.M.Barzdin, Some rules of inductive inference and their use for program synthesis, *Proc. IFIP 9th World Congress*, Paris, 1983, R.E.A.Mason(ed.), North-Holland 1983.

[BBD 81]

F.L.Bauer, M.Broy, W.Dosch, R.Gnatz, B.Krieg-Brueckner, A.Laut, M.Luckmann, T.Matzner, B.Möller, H.Partsch, P.Pepper, K.Samelson, R.Steinbruggen, M.Wirsing, H.Wössner, Report on a wide spectrum language for program specification and development. *T.U.M. Report 18104*, Institut für Informatik, Tech. Univ. München.

[BBG 84]

M.Bidoit, B.Biebow, M.C.Gaudel, C.Gresse, G.Guiho, Exception Handling: Formal Specification and Systematic Program Construction, Nato ASI Series vol. F8, *Program Transformation and Programming Environments*, Springer-Verlag 1984.

[BeU 86]

B.Belkhouche, J.E.Urban, Direct Implementation of Abstract Data Types from Abstract Specifications, *I.E.E.E. Trans. on Soft. Eng.*, vol. SE-12, no. 5, May 1986.

[BMP 86]

M.Broy, B.Möller, P.Pepper, M.Wirsing, Algebraic Implementations Preserve Program Correctness, *Science of Computer Programming* 7 1986.

[BoM 79]

R.S.Boyer, J.S.Moore, *A Computational Logic*, Academic Press, 1979.

[BOR 81]

U.Bartels, W.Olthoff, P.Raulefs, APE: An Expert System for Automatic Programming from Abstract Specifications of Data Types and Algorithms. *MEMO SEKI-BN-81-01*, Institut für Informatik, Universitaet Kaiserslautern, 1981.

[BoW 80]

T.L.Booth, C.A.Wiecek, Performance Abstract Data Types as a Tool in Software Performance Analysis and Design, *I.E.E.E. Transactions on Software Engineering*, vol. SE-6, no. 2, March 1980.

[Bro 84]

M.Broy, Algebraic Methods for Program Construction: The Project CIP, pp. 199-222, Nato ASI Series vol. F8, *Program Transformation and Programming Environments*, Springer-Verlag 1984.

[BrW 80]

M.Broy, M.Wirsing, Programming Languages as Abstract Data Types, in *Proc. of 5th CAAP*, Lille, M. Dauchet (ed.), 1980.

[BrW 81]

M.Broy, M.Wirsing, On the Algebraic Extensions of Abstract Data Types, Proceedings of Formalisation of Programming Concepts, Peniscola, April 1981, *Lecture Notes in Computer Science* 107, Springer-Verlag 1981.

[BrW 82]

M.Broy, M.Wirsing, Partial Abstract Data Types, *Acta Informatica*, 18, no. 1, 1982.

[BuD 77]

R.M.Burstall, J.Darlington, A Transformation System for Developing Recursive Programs, *J.A.C.M.* vol. 1,no. 24, January 1977.

[BuG 81]

R.M.Burstall, J.A.Goguen, An informal introduction to specifications using Clear, in *The Correctness Problems in Computer Science,* (eds. Boyer,Moore), Academic Press 1981.

[CKS 87]

C.Choppy, S.Kaplan, M.Soria, Algorithmic Complexity of Term Rewriting Systems, *Report no. 344*, Lab. de Recherche en Informatique, Centre d'Orsay, Bat. 490, Université de Paris-Sud.

[Dar 82]

J.Darlington, Program Transformation, in *Functional Programming and its Applications, An Advanced Course*, Darlington et al (Eds.), Cambridge University Press, 1982.

[Der 85]

N.Dershowitz, Orderings for Term-Rewriting Systems, *Theoretical Computer Science* 17, 1982.

[Dic 87]

A.J.J.Dick, Equational Reasoning and Rewrite Systems on a Lattice of Types, *Ph.D. Thesis*, Dept. of Computing, Imperial College, London 1987.

[DIm 69]

M.E.D'Imperio, Data Structures and their Representation in Storage, *Annual Review in Automatic Programming*, 5, Halpern, Shaw (Eds.), Pergamon Press Ltd., 1969.

[Ear 71]

J.Earley, Toward an Understanding of Data Structures, *C.A.C.M.*, vol. 14, no. 10, 1971.

[EhM 85]

H.Ehrig, B.Mahr, *Fundamentals of Algebraic Specification 1*, EATCS Monographs on Theoretical Computer Science, Springer-Verlag 1985.

[EKM 82]

H.Ehrig, H.-J.Kreowski, B. Mahr, P.Padawitz, Algebraic Implementation of Abstract Data Types, *Theoretical Computer Science*, North-Holland Publ. Co. 1982.

[Ehr 82]

H.-D. Ehrich, On the Theory of Specification, Implementation, and Parameterization of Abstract Data Types, *J.A.C.M.*, vol. 29, no. 1, January 1982.

[EKR 80]

H.Eigemeier, Ch.Knabe, P.Raulefs, K.Tramer, An Expert System for Automatic Coding of Abstract Data Type Specifications, *Informatik Fachberichte Jahrestagung,* vol. 33, Springer-Verlag, 1980.

[End 77]

H.B.Enderton, *Elements of Set Theory*, Academic Press 1977.

[FGJ 85]

K.Futatsugi, J.A.Goguen, J-P.Jouannaud, J.Meseguer, Principles of OBJ2, *Proceedings of Principles of Programming Languages*, A.C.M.,1985.

[FoG 84]

R.Forgaard, J.V.Guttag, REVE: A Term Rewriting System Generator with Failure-Resistant Knuth-Bendix, Laboratory for Computer Science, MIT, 1984.

[Fri 86]

L.Fribourg, A Strong Restriction of the Inductive Completion Procedure, in Proc. of 13th ICALP, Rennes, L. Kott (ed.), *Lecture Notes in Computer Science* 226, Springer-Verlag 1986.

[Gan 87]

H.Ganzinger, Ground Term Confluence in Parametric Conditional Equational Specification, in Proc. of STACS 87, Passau, *Lecture Notes in Computer Science* 247, Springer-Verlag 1987.

[GeM 86]

N.Gehani, A.D.McGettrick (Eds.), *Software Specification Techniques*, Addison-Wesley 1986.

[GHM 78a]

J.V.Guttag, E.Horowitz, D.R.Musser, The Design of Data Type Specifications, Number 4, Chapter 4, *Current Trends in Computer Science*, 1978.

[GHM 78b]

J.V.Guttag, E.Horowitz, D.R.Musser, Abstract Data Types and Software Validation, *C.A.C.M.* vol. 21, no. 12, Dec. 1978.

[GJM 85]

J.A.Goguen, J-P.Jouannaud, J.Meseguer, Operational Semantics for Order-Sorted Algebra, Proceedings of I.C.A.L.P. 1985, *Lecture Notes in Computer Science* 194, Springer-Verlag.

[GoB 83]

J.A.Goguen, R.M.Burstall, Introducing Institutions, Proceedings of Logics of Programs, *Lecture Notes in Computer Science* 164, Springer-Verlag,1983.

[Gog 87]

M.Gogolla, On Parametric Algebraic Specifications with Clean Error Handling, TAPSOFT 87 Pisa, Italy, *Lecture Notes in Computer Science* vol. 249, Springer-Verlag 1987.

[GoM 83]

J.A.Goguen, J.Meseguer, Initiality, induction, computability, Chapter 14, *Algebraic Methods in Semantics*, Nivat (ed.), C.U.P. 1983.

[GoM 87a]

J.A.Goguen, J.Meseguer, Models and Equality for Logical Programming, TAPSOFT 87, Pisa, Italy, *Lecture Notes in Computer Science* 249, Springer-Verlag, 1987.

[GoM 87b]

J.A.Goguen, J.Meseguer, An Order-Sorted Algebra Approach to the Constructors and Selectors Problem, Draft Manuscript, To appear in *Topics in Computer Science* ,1987.

[GoP 81]

J.A.Goguen, K.Parsaye-Ghomi, Algebraic Denotational Semantics using Parameterised Abstract Modules, in Proc. of Formalization of Programming Concepts, Peniscola, April 1981, *Lecture Notes in Computer Science* 107, Springer-Verlag, 1981.

[GuH 78]

J.V.Guttag, J.J.Horning, The Algebraic Specification of Abstract Data Types, *Acta Informatica* 10, 1978.

[Hal 67]

P.R. Halmos, *Naive Set Theory*, D.Van Nostrand Company, Inc., 1967.

[Har 69]

F.Harary, *Graph Theory*, Addison-Wesley, 1969.

[Hay 87]

I. Hayes (Ed.), *Specification Case Studies*, Prentice-Hall, 1987.

[Hoa 72]

C.A.R. Hoare, Proof of Correctness of Data Representations, *Acta Informatica* 1, 1972.

[HoH 86]

C.A.R. Hoare, Ji Feng He, The Weakest Prespecification, *Fundamenta Informatica* IX, 1986.

[HsS 85]

J.Hsiang, M.K.Srivas, A PROLOG Environment for Developing and Reasoning about Data Types, TAPSOFT 85, Berlin, FRG, *Lecture Notes in Computer Science 186*, Springer-Verlag, 1985.

[HuH 82]

G.Huet, J.Hullot, Proofs in Equational Theories with Constructors, *Journal of Computer and Systems Sciences*, 1982.

[HuO 80]

G.Huet, D.Oppen, Equations and Rewrite Rules: A Survey, in *Formal Language Theory: Perspectives and Open Problems*, R. Book (ed.), pp 349-405, Academic Press, New York, 1980.

[JaT 87]

K.PJantke, M.Thomas, A Note on Inductive Inference for Solving Divergence in Knuth-Bendix Completion, preliminary version in *Stirling University Technical Report*, no. 41 (to appear in *Proc. of 5th Workshop on Specification of Abstract Data Types*, Gullane, Sept.1987, forthcoming).

[JoK 87]

J.-P. Jouannaud, E.Kounalis, Automatic Proofs by Induction in Theories without Constructors, Revised Manuscript, preliminary version in *Proc. Ist IEEE Symposium on Logic in Computer Science*, 1986.

[Jon 86]

C.B. Jones, *Systematic Software Delopment Using VDM*, Prentice-Hall,1986.

[JøS 86]

U. Jørring, W. Scherlis, Deriving and Using Destructive Data Types, in *IFIP TC2 Working Conference on Program Specification and Transformation*, North-Holland 1986.

[KaB 81]

E.Kant, D.R.Barstow, The Refinement Paradigm: The Interaction of Coding and Efficiency Knowledge in Program Synthesis, *IEEE Trans. on Soft. Eng.*, vol SE-7, no.5, Sept. 1981.

[KaM 87]

D.Kapur, D.R.Musser, Proof by Consistency, *Artificial Intelligence* 31, 1987.

[KaS 85]

D.Kapur, M.K. Srivas, A Rewrite Rule Based Approach for Synthesising Abstract Data Types, TAPSOFT Berlin 1985, *Lecture Notes in Computer Science* 185, Springer-Verlag 1985.

[KnB 70]

D.E.Knuth, P.B.Bendix, Simple Word Problems in Universal Algebras, *Computational Algebra*, J.Leach(ed.), Pergamon Press 1970.

[Lan 87]

S.Lange, A Decidability Problem of Church-Rosser Specifications for Program Synthesis, Proceedings of Analogical and Inductive Inference Workshop, Wendisch-Rietz, GDR, October 1986, *Lecture Notes in Computer Science* 265, Springer-Verlag 1987.

[Low 78]

J.R.Low, Automatic Data Structure Selection: An Example and Overview, *C.A.C.M.* vol. 21, no. 5, May 1978.

[Met 86]

Meteor Project, Esprit Project no. 432.

[Mar 86]

J.Martin, *Data Types and Data Structures*, Prentice Hall 1986.

[Mit 87]

B.Mitchell, Some Aspects of Induction, Technical Report, Dept. Computer Science, Manchester University, Manchester 1987.

[Moi 82]

A.Moitra, Direct Implementation of Algebraic Specification of Abstract Data Types, *IEEE Trans. on Soft. Eng.*, vol SE-8, no. 2, Jan. 1982.

[Mus 80a]

D.R.Musser, On Proving Inductive Properties of Abstract Data Types, *Proceedings of the seventh ACM Symposium on Principles of Programming Languages*, Las Vegas, Nevada, Jan. 1980.

[Mus 80b]

D.R.Musser, Abstract Data Type Specification in the Affirm System, *IEEE Trans. Soft. Eng.*,vol SE-6, no. 1, Jan. 80.

[Möl 87]

B. Möller, Algebraic Specifications with Higher-Order Operators, in *Proc. IFIP TC2 Working Conference on Program Specification and Transformation*, Bad Tölz, April 1986, L.Meertens (Ed.), North-Holland, 1987.

[Olt 85]

W.Olthoff, Private Communication, 1985.

[Pad 85]

P.Padawitz, Parameter Preserving Data Type Specifications, in Proc. TAPSOFT '85 Berlin, FRG, March 1985, *Lecture Notes in Computer Science*, vol. 185, Springer-Verlag 1985.

[PaK 83]

R.Paige, S.Koenig, Finite Differencing of Computable Expressions, *ACM TOPLAS* 4, 3, July 1983.

[PaK 87]

W.Pase,S.Kromodimoeljo, NEVER An Interactive Theorem Prover, Technical Report, I.P.Sharp Associates Ltd., 265 Carling Ave., Ottawa, Canada 1987.

[Pau 85]

E.Paul, On Solving the Equality Problem in Theories Defined by Horn Clauses, Proceedings of the European Conference on Computer Algebra, Linz, April 1985, *Lecture Notes in Computer Science* vol. 204, Springer-Verlag 1985.

[Per 86]

G.Persch, Automating the Transformational Development of Software, Meteor Project Technical Report, GMD Forshungsstell an der Univ. Karlsruhe, 1986.

[PeS 81]

G.E.Peterson, M.E.Stickel, Complete Sets of Reductions for Some Equational Theories, *J.A.C.M.*, vol. 28, no. 2, 1981.

[PeV 78]

T.H.C.Pequeno, P.A.S.Veloso, Do Not Write More Axioms Than You Have To, *Proceedings of International Computer Symposium*, vol. 1, 1978.

[Pro 85]

Prospectra Project Summary, ESPRIT Project no. 390.

[RoK 77]

S.J.Rosenschein, S.M.Katz, Selection of Representations for Data Structures, *Sigplan Notices*, vol. 12, no. 8, Aug. 1977.

[Ros 71]

A.L.Rosenberg, Data Graphs and Addressing Schemes, *Jnl of Computer and System Sciences*, 5,1971.

[RoT 78]

L.A.Rowe, F.M.Tong, Automating the Selection of Implementation Structures, *IEEE Trans. on Soft. Eng.*, vol SE-4, no. 6, Nov 1978.

[San 84]

D.Sannella, A Set-Theoretic Semantics for Clear, *Acta Informatica*, 21, 1984.

[SaW 82]

D.Sannella, M.Wirsing, Implementation of parameterised specifications, Proc. of 9th ICALP, *Lecture Notes in Computer Science*, vol. 140, Springer-Verlag, 1982.

[Sch 85]

W.Scherlis, Adapting Abstract Data Types, Carnegie Mellon University, *1985.*

[Sch 86]

W.Scherlis, Abstract Data Types, Specialization and Program Reuse, in *Intl. Workshop on Advanced Programming Environments*, Springer-Verlag, 1986.

[ShS 74]

B.Shneiderman, P.Schuermann, Structured Data Structures, *C.A.C.M.*, vol.17, no.10, Oct. 1974.

[SSS 81]

E.Schonberg, J.T.Schwartz, M.Sharir, An Automatic Selection of Data Representations in SETL Programs, *A.C.M. T.o.P.L.a.S.*, vol. 3, no. 2, Apr. 1981.

[Sum 75]

P.D.Summers, A Methodology for LISP Program Construction from Examples, in Proc. of 3rd. A.C.M. P.o.P.L. , 1975, Atlanta, U.S.A., A.C.M. 1976.

[Tho 87]

M.Thomas, Implementing Algebraically Specified Abstract Data Types in an Imperative Programming Language, TAPSOFT 87 Pisa, Italy, *Lecture Notes in Computer Science* vol. 250, Springer-Verlag 1987.

[Tho 86]

S.J.Thompson, Laws in Miranda, *Proceedings of 1986 ACM Conference on LISP and Functional Programming*, ACM, 1986.

[Wir 86]

M.Wirsing, Structured Algebraic Specifications: A Kernel Language, *Theoretical Computer Science*, 42, 1986.

[WPP 83]

M.Wirsing, P.Pepper, H.Partsch, M.Broy, On Hierarchies of Abstract Data Types, *Acta Informatica*, 20, 1983.